AGARD-CP-503

AGARD-CP-503

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT
7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD CONFERENCE PROCEEDINGS 503

Software for Guidance and Control

(Les Logiciels de Guidage et de Pilotage)

NORTH ATLANTIC TREATY ORGANIZATION

Distribution and Availability on Back Cover

AGARD-CP-503

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT
7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD CONFERENCE PROCEEDINGS 503

Software for Guidance and Control

(Les Logiciels de Guidage et de Pilotage)

Papers presented at the Guidance and Control Panel 52nd Symposium
held at the Helexpo, Thessaloniki, Greece, from 7th May to 10th May 1991.

North Atlantic Treaty Organization
Organisation du Traité de l'Atlantique Nord

The Mission of AGARD

According to its Charter, the mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

- Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community;
- Providing scientific and technical advice and assistance to the Military Committee in the field of aerospace research and development (with particular regard to its military application);
- Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;
- Improving the co-operation among member nations in aerospace research and development;
- Exchange of scientific and technical information;
- Providing assistance to member nations for the purpose of increasing their scientific and technical potential;
- Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

Published September 1991

ISBN 92-835-0629-4

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

Theme

Software is of increasing importance in guidance and control systems and indeed in many cases is the pacing item in development. Guidance and control software, while embracing a wide range of software, has emphases which include high integrity considerations, hard real-time constraints, the implications of a still evolving hardware and systems architecture, and the need to meet delivery schedules with high productivity under the constraints of onerous customer requirements for documentation and visibility, and in the light of strong defence and air worthiness standards and requirements. Time schedules are frequently short since much guidance and control software is required early in the flight testing, and typically software development is undertaken in the context of still evolving requirements and developing programme phases.

The software climate in which this takes place is one in which there is a general trend towards high level languages, integration of support tools, introduction of mathematical formalisms into the design and verification processes, control of software sizing and better cost estimating, and frequently a rapid turnover of programmers.

There is often a wide gap between concept and practice, and organizations will succeed which can bridge the gap effectively, bringing modern methodologies, well supported by software tools, to bear on the problem and understanding how to apply these methodologies and use the tools.

To assist this understanding the symposium covered general requirements on the software, software requirements capture, design methods and support environments for real-time software, coding techniques, and verification validation and certification.

Thème

Les logiciels revêtent de plus en plus d'importance dans les systèmes de guidage et de pilotage. En effet, le logiciel est souvent l'élément critique pour le développement des systèmes.

Bien qu'il existe une large gamme de logiciels de guidage et de pilotage, l'accent est mis principalement sur les considérations suivantes: la haute intégrité, les contraintes temps réel du matériel, les conséquences de l'évolution permanente des architectures systèmes et matériel, le respect des délais de livraison pour des volumes de production élevés, la demande onéreuse de documentation de la part du client, les contraintes d'intelligibilité du logiciel, et la rigueur des spécifications et normes militaires et aéronautiques. Les délais sont souvent courts, puisque bon nombre des logiciels de guidage et de pilotage sont demandés dès la première phase des essais en vol; typiquement, le logiciel est crée pendant que les besoins continuent à évoluer dans le contexte des différentes phases évolutives du programme.

L'environnement logiciel de ce processus est caractérisé par les langages évolués, l'intégration des outils de développement, l'emploi de formalismes mathématiques dans les méthodes de conception et de vérification, le contrôle du dimensionnement des logiciels, la recherche d'une meilleure estimation des coûts et le renouvellement fréquent des programmeurs.

Il existe souvent un grand pas à franchir pour passer du concept à la pratique. Les organisations qui réussiront à l'avenir seront celles qui sauront franchir ce pas de façon efficace, en se servant de méthodologies modernes, bien appuyées par des outils de développement, et qui auront compris l'application de ces méthodologies et la mise en oeuvre de ces outils.

Afin de faciliter cette compréhension, le symposium a examiné les sujets suivants: conditions générales requises pour les logiciels, élaboration des spécifications, méthodes de conception et environnements de soutien pour les logiciels temps reel, techniques de codage, et vérification, validation et certification.

Guidance and Control Panel

Chairman: Dr E B. Stear
Corporate Vice President
Technology Assessment
The Boeing Company
PO Box 3707
Mail Stop 13—43
Seattle, WA 98124-2207
United States

Deputy Chairman: Mr S. Leek
Scientific Adviser
British Aerospace (Dynamics) Ltd, PB 256
PO Box 19
Six Hills Way
Stevenage
Herts SG1 2DA
United Kingdom

TECHNICAL PROGRAMME COMMITTEE

Chairman:	Professor J T. Shepherd	UK
Members:	Dr André Benoit	BE
	Mr B. Jaeger	FR
	Mr U.K Krogmann	GE
	Lt F. Hatzivasilhou	GR
	Mr K.A. Helps	UK
	Mr S. Haaland	US
	Dr J. Niemela	US
	Dr E. Zimet	US

HOST PANEL COORDINATOR

Lt Fivos Hatzivasilhou
Hellenic Air Force
Technology Center (KFTA)
Terpsithea Post Office
16501 Glyfada, Athens
Greece

PANEL EXECUTIVE

Commandant M Mouhamad, FAF

Mail from Europe:
AGARD—OTAN
Attn: GCP Executive
7, rue Ancelle
F-92200 Neuilly sur Seine
France

Mail from US and Canada:
AGARD—NATO
Attn: GCP Executive
APO AE 09777

Tel. 33(1) 47 38 57 80
Telex 610176 (F)
Telefax. 33(1) 47 38 57 99

ACKNOWLEDGEMENTS/REMERCIEMENTS

The Panel wishes to express its thanks to the Greek National Delegates to AGARD for the invitation to hold this meeting in their country and for the facilities and personnel which made the meeting possible.

Le Panel tient a remercier les Délegués Nationaux de la Grece près l'AGARD de leur invitation à tenir cette reunion dans leur pays et de la mise à disposition de personnel et des installations necessaires

Contents

Reference

SESSION VII: AUTOMATED SOFTWARE GENERATION APPROACHES*
(Final Report from Working Group 10)
Chairman: Dr E.B.Stear (US)

* As papers presented in this session represent the final work of Working Group 10, Papers 26 to 29 are not included in this Conference Proceedings. The final report of Working Group 10 will be published as an Advisory Report (AR-292) in 1991/92

A Survey of Available Tools and Methods for Software Requirements Capture and Design

by
D.J.Thewlis
GEC Marconi Ltd..
The Grove, Warren Lane,
Stanmore,Middlesex,
HA7 4LY,
England.

1. Introduction

In this paper I discuss the contribution to software development of the tools and methods currently available to assist with the early part of the software development life cycle - that is tools and methods for Requirements Capture and design. For all parts of the life cycle the requirement for Quality, that is Quality Assurance and Quality Control is fundamental. If it is lacking then, at worst, the software will never achieve a deliverable state, if delivered it will not work well and will probably not be maintainable. The word 'Quality' is used in an informal sense, this paper is not about Quality Management or Quality Control so these requirements are not discussed in detail; nevertheless, the case is made that the tools and methods used determine the achievable quality of the delivered system. There is a sense in which the output from these tools is part of the delivered system. In general, the quality requirements for a large or complex system are greater than those for a small or simple system, so the tools and methods now available make it feasible to build systems of a size or complexity which would be impossible without them. The newer tools which are emerging should enable the inevitable demand for even bigger systems to be met.

Although the methods emerged before tools were created to instantiate them, the words 'method' and 'tool' are effectively interchangeable. Taking a simple example from a related area; critical path analysis of a network is a project planning method. It would be practically impossible to apply this method to even quite a small project without the use of a network analysis programme - the tool. To the project manager his project management method is the tool which he uses. It is so with software design methods and tools. Methods on their own are of little use: fortunately, we now have tools.

My company has used the MASCOT method and is experienced in the use of the following tools:

Teamwork	(Yourdon)
Software through Pictures	(Yourdon)
Speedbuilder	(Jackson)
PDF	(Jackson)

We are experimenting with the use of a Hierarchical Object Oriental Design (HOOD) tool, so in this paper the discussion of methods and their contribution to software development comes from this experience.

2. Requirements and Design.

Current Dogma is that we collect together a complete specification of the customers requirements before embarking on design. Further, we should ensure that this specification is not contaminated by design or implementation considerations. The arguments for this are powerful. One such is that: if we approach requirement capture with preconceived ideas on implementation; we shall try to make the requirements fit the design rather than the design fit the requirements. At worst we fail to capture important parts of the requirement, at best we have a less than optimal design. Ward and Mellor [1], amongst others, have developed this argument. They argue that technology is now so developed that technology limits are now irrelevant and we can, so should, deliver exactly what the customer requires.

That technology limits are irrelevant is debatable and is covered in section 3. Equally debatable is whether it is possible to acquire the full requirement prior to starting the design. In practice the design process uncovers so many questions about the requirement that it becomes impossible to separate the processes of requirement capture and (high level) design. The Starts Guide [2] to software methods and tools accepts this, treating both requirements and design in the same chapter and reporting that many of the tools they cover apply to both parts of the life cycle.

Computers have existed for about forty years and have been widely used for twenty. There must be few situations left where computers are being used for the first time. Most projects are to produce a system which is in some ways better than an already existing system. The customers perception of his requirements is coloured by his knowledge of current implementations. And considerations of cost and risk reduction constrain the supplier to use existing ideas, designs and code wherever possible. Thus the ideal, establishing a requirement free from implementation considerations, is rarely achievable. Most writers on methods and tools assume the ideal.

There are few projects where the requirements established at the beginning of development are still

valid at the time of delivery. Requirements change with time, so the methods and tools have to be robust enough to cope with changes in the requirement. Fortunately, the tools which have emerged in the last few years do seem to work well in the real world.

Although the methods discussed have features which apply particularly to the initial requirements capture, such features are not discussed in this paper as they are not relevant to the main argument and, in the author's opinion, of less importance than the facilities which assist design and the control of design.

3. The Software Problem.

Computer hardware performance increases at a formidable rate Some years ago the IBM UK Research Director stated that processors were getting "better" at a rate of 45% compound per year. Memory and data storage were not doing so well; for them the rate was 25% compound per year "Better" means almost any ratio which is likely to interest system developers. that is MIPs or bytes as the numerator and such things as price, size or power consumption as the denominator. These rates have applied since the early days of computing and are likely to apply for the foreseeable future

It has long been the hope of software developers that some of this 'surplus' power could be used to aid the task of software engineering. The statement that the industry has developed to an extent that we are problem limited rather than technology limited in reference [1] is one in a sequence of similar statements going back to the early days of computing In practice, the technology limits continue to apply; the demands for speed and functionality which systems engineers are placing on computer system designers are increasing as quickly as the hardware technology improves

It therefore seems likely that the size of system that developers are invited to construct will increase at a rate similar to that at which hardware improves. Experience within our organisation supports this Examination of similar projects, one started four years later than the other, showed that each project had a similar size team working for about the same time, but the later project produced 2½ times more software than the earlier project The methods and tools used by developers have to keep pace with the increasing size of software projects.

4. Quality.

A few men with buckets and shovels can mix concrete and lay a garden path or the foundations for a garage Perhaps a large number of men with the same tools could mix and lay the concrete for a bridge or a nuclear reactor shield. No doubt problems, such as delivering the concrete to the right place at the right time, could be solved, so it is in principle feasible In practice, the bridge would fall down and the reactor shield leak because the quality and consistency of concrete sufficient for a garden path would not be sufficient for a larger structure.

It is so with software Although many large systems were produced in assembly code; even more were attempted but never used because their quality was too low The size and sophistication of the systems we can build is constrained by the tools we have available.

Tools or methods for producing software, potentially have two aspects.

> Chunking. that is the ability to 'see' the software or a part of the software as a small enough number of chunks to be understandable.
>
> Complexity: that is the connectivity between the chunks is of low enough complexity to be understandable

High level languages, eg FORTRAN 2, were the first tools, they enabled a number of machine code statements to be created by one high level language instruction. Further chunking was achieved by the concept of a subroutine. For mathematical software that, together with some ad hoc rules to control complexity, has been sufficient to produce large systems of high quality. For other types of software this was necessary but not sufficient. New high level languages such as PASCAL and, following that, Ada were devised. These give more chunking than FORTRAN permits with data structure and control complexity by the introduction of composite data types which can be manipulated as a whole and encouraging, almost enforcing, structured programming as defined by Dijkstra The methods and tools discussed in this paper take these concepts further

There are a number of methods which have evolved over the last 10 years They are proven, in that numerous systems have been developed using them, and they are supported by tools, available in the market place, which can themselves be regarded as proven

After high level languages the next step was modular programming, initially an undefined term until Stevens, Myers and Constantine [3] and Parnas [4] developed the following qualities which defined a good module.

> High cohesion - related activities are grouped together.
>
> Low coupling - minimum data transfer between modules
>
> Data hiding - modules require no (or little) knowledge of how data is structured in other modules

The current generation of methods and tools support the concepts of low coupling and data hiding. They are not enforced, so it is possible to produce a poor design; but their use is encouraged. The design is made visible in the diagrams created by the tools, so it is easy to inspect the design to discover whether it is of sufficiently high quality.

The Yourdon and the Mascot methods assume a 'top down' design process. They are, for reasons discussed in paragraph 8, less good at supporting the concept of high cohesion.

In addition to the aid they give in supporting good design principles, the tools have two other function which affect quality: they record the current state of the design and assist with reviewing the design. They present the design in a way that is easy to understand and update when the design has to change because of requirement change or implementation considerations.

For many companies in our group the improvement in documentation and design control brought by the design tool is the main reason for using the tool. We may or may not design 'top down': we do review 'top down'.

Since these methods were defined ideas on system development have evolved. These develop further the idea of a module. They assume that a system can be decomposed into a number of objects. Object is a technical term. It means a piece of software which delivers a service when requested by other objects but whose internal structure and data is hidden from other objects. These Object Oriented Design methods, in effect, enforce the creation of modules which are good in the sense described above.

5. Current Design Methods and Tools.

Three methods, Yourdon, Jackson System Design and Mascot seem the most popular amongst designers of real time systems. The main part of each system is a diagram consisting of boxes joined by arrows thus:-

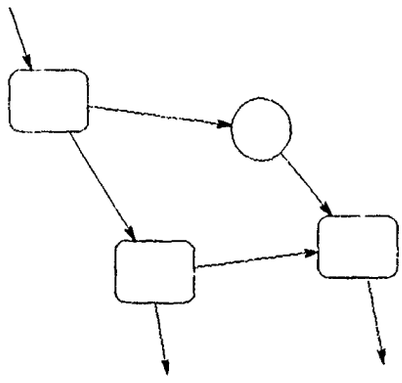


Figure 1

Boxes are places in which data is transformed. Arrows are routes along which data or signals travel. This is called a data flow diagram. Each method has its own vocabulary and systems. Some of the boxes have a specialised function and different types of information flow are marked by some device such as double arrows or dotted lines.

Two of the methods have a significant feature: there is one type of box which is recursively defined. That is, it can itself contain boxes and arrows. This gives great chunking power. A large system can be described by few (less than 10) boxes with arrows. Each box, itself containing further boxes and arrows and so on to whatever level of detail is necessary. The Jackson tools tend not to support this feature. Jackson is a 'Middle out' rather than a top down method.

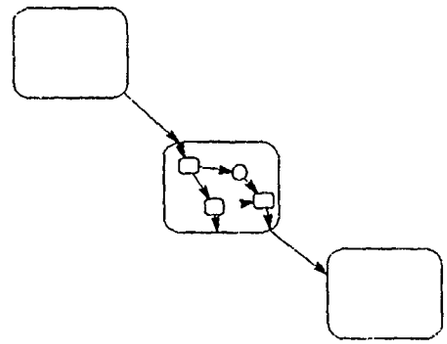


Figure 2

The methods support good design principles. With one exception, discussed below, no box can get at the data belonging to another box, except via the arrowed routes, using the appropriate syntax for that route. Thus, the Parnas [4] concept of 'information hiding' is enforced to the extent that no box can change the data held by another box. Although not enforced, the principle of low coupling is encouraged. A design failing to meet this criteria would have too many arrows, and so be recognisable as a bad design. Composite data types are supported by all methods. Only data of pre defined types can pass along the arrowed routes. Application of these methods using pencil and paper would be difficult, tools are necessary. They maintain a data base of the design, check that each arrow both starts and finishes in an appropriate place, and preserve consistency. A most important feature of the tools is that when a change is made to the design, they bring to the attention of the designer the consequential design changes which are required.

A vital feature of all three methods is that they all go beyond the Von Neuman model of computing. The boxes operate asynchronously. That is each box could, in principle, be a separate processor Thus, in the early part of the work, the design is

independent of the detail of the hardware on which the system will run. If, when implemented, the system runs too slowly, the same design can be used to create a system which uses more or different processors.

In the Yourdon Method this concept of "implementation free design" is taken further. A designer using Yourdon is instructed to build his data flow diagram without considering how data will be transferred or processed. For example, a data flow might well be "reactor temperature and pressure". In this form the data flow diagram is called the essential model. The next stage in the design is to refine the essential model to create the implementation model. At this stage data types are made explicit - in the above example reactor temperature might be defined as a three digit decimal number. The transfer medium and protocol would also be defined. This distinction between essential and implementation models is a feature of the Yourdon Method. The other methods could be used that way. In some, perhaps most, situations the distinction is valuable - details of the implementation can be changed without affecting the essential model and the designer is concentrating on one kind of thing at a time. In the essential model he is working on what has to be done; in the implementation model he concentrates on how it will be done.

6. Full Constituents of a design method.

There are three main elements

> A description of how the system interfaces with its environment.
>
> The design diagrams - (data flow diagrams)
>
> Translation of the design into a computer program.

Producing the design diagrams is the creative part of design and is discussed below The Jackson method for the other two activities is the easiest to describe. Taking the translation into code first. The method is called Jackson System Programming, [7]. It is based on the Dijkstra concepts of structured programming; sequence, iteration and selection are the only control structures permitted.

It uses boxes connected as follows. They form a tree. that is each of the boxes A, B and C can themselves have boxes beneath them to whatever depth is appropriate.

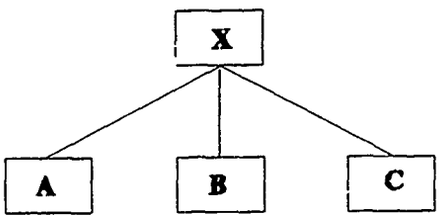


Figure 3

This means X is A followed by B followed by C.

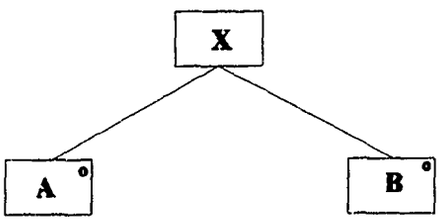


Figure 4

This means X is either A or B

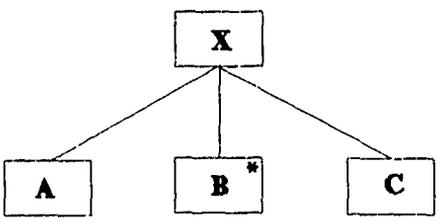


Figure 5

And this is X is A then a number of Bs followed by C.

The program code is then created by following through the tree in a logical order. The other methods go through a similar process for code JSP, Jackson System Programming, was the precursor to JSD. A major step in the creation of JSD was the recognition that the systems' interaction with its environment could be described using a similar structure to that of JSP The data which a computer system processes can be described using the concepts of sequence, iteration and selection. For example, a data sequence could be an 'A' followed by a number of 'B's, then a 'C' or an 'A' followed by either a 'B' or a 'C', then a D. The diagram used to describe the data input to a system are essentially the same as those used to describe processing at the code level Although they use a rather different method from JSP to describe code, users of the Yourdon method are tending to use the JSD method of describing data inputs from a systems' environment.

7. Differences between the methods.

Differences between the methods are apparent in the way they handle information flow between processor boxes. The essentials are summarised below.

7.1 Mascot.

Of the three methods the communications are most well defined in Mascot. Necessarily so, for Mascot originally had run time environment as well as being a design method. No activity box can communicate directly with any other activity box. Communication is via an IDA, intercommunication data area. Originally there were two of these, channel and pool, the current version of MASCOT, MASCOT 3 permits more complex IDAs.

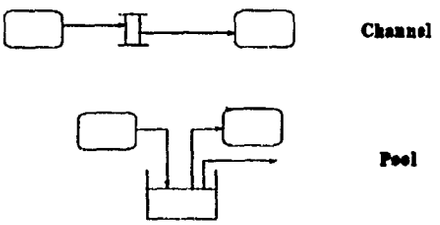


Figure 6

A channel is a buffered, first in, first out store. Such facilities are common to other methods. This enables activities to progress asynchronously. The sending activity can send it down to the channel then continue with its work. The receiving activity collects the data from the channel when it is ready to do so.

A pool is more complex. It always contains some data. A number of activities can update the information held by the pool. A number of activities can read the data in the pool. The difference between a channel and a pool is that, data is destroyed when read in a channel; data remains in a pool until overwritten. A full description of Mascot is in reference [5].

7.2 Jackson System Design.

Like MASCOT, JSD has communication methods which enable activities to progress asynchronously. There are two methods. One, called a data stream, is similar to a MASCOT channel. The other enables one activity to inspect (but not write to) the state vector of another activity. This is the same sort of facility as a MASCOT pool. A purist may argue that it seems less safe but the facility can be used to create an activity which duplicates the action of a MASCOT pool.

With a data stream the initiator is the activity supplying the data. The activity requiring data initiates a state vector inspection. The Jackson methods, both for design and programming are in reference [6].

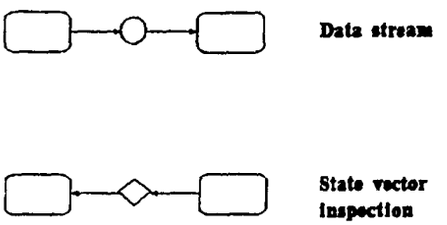


Figure 7

7.3 Yourdon.

This method has a philosophy different from the other two methods. The Yourdon method distinguishes between different kinds of information. There are two kinds, data and signals. Signals announce that an event has taken place or issue a command. Yourdon has one type of data store, which is more like a MASCOT pool than a channel, although the effect of a channel can be created using control signals.

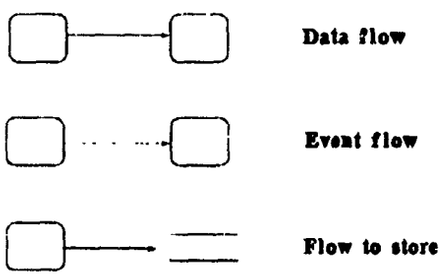


Figure 8

The facility of separating control signals from data is powerful. It is enhanced by the provision of finite state machine models to sort out the consequences of complex interactions between control signals. For some types of real time systems this facility alone makes Yourdon the method of choice.

8. Weakness of these methods.

A top down design methodology assists the designer in achieving two of the three criteria of good design, data hiding and low coupling. It contributes less towards the aim of high cohesion.

There is little assistance to the designer in identifying and implementing modules which provide Common services. Especially so on a large project when, following the top level design, the rest of the work is partitioned amongst a number of different designers. A middle out approach, for the reasons given below, enables a balance to be struck between low coupling and high cohesion, it does not neccessarily achieve both.

One of our companies, which uses the Yourdon method, finds the methods' poor support for 'high cohesion' so much of a problem that they have drastically revised the way they operate the method. They make little use of the tool as a design aid because, having tried to use it that way, they found that they were creating unsatisfactory designs. The problem seemed to be that the decomposition of the system coming from the top down methodology was not resulting in a good system. The top level diagrams when first produced seemed satisfactory but, as the design proceeded to lower levels, and the designers understanding of what was necessary improved, the design became less satisfactory. Either the number of data flows in the data flow diagrams increased; or a lot of similar things were done in different parts of the system. Either high cohesion or low coupling could be achieved, but not both. For the kinds of system this company is producing, large radar systems, they find that a better design route is to identify the low level modules which will be required. These are then put into groups, then groups into larger groups and so on. Although the Yourdon tool is not used as a design method; it is used to record the design, maintain it as the requirement evolves and to support design review.

This is the experience of one company in our group. Some other companies are following their lead. Other companies find the top down approach satisfactory for the kind of software they produce, so use the tools both as a design aid and a method of recording the design.

The newer methods, Object Oriented Design, preserve the advantages of current methods whilst encouraging high cohesion. They are discussed below.

9. Objected Oriented Methods.

Although not entirely a new idea, their precursor - SIMULA emerged in the 1960s, Object Oriented Methods have surged in popularity in recent years. Smalltalk, followed by C + + have contributed at the programming level; so has Ada which has many of the features of an object oriented language.

The Object Oriented approach considers a system to be a set of interacting objects. Each object provides a service or services to other objects. Objects request a service of other objects by sending a message to that object. A message consists of a request for a service together with any necessary data. Objects have access to one another only via messages. This is the heart of the matter. Reference [8] contains a full description of object oriented programming including those aspects not covered in this paper. These programming ideas have been carried into the requirement capture and design part of the life cycle. In Europe the development of HOOD, hierarchical object oriented design, by the European Space Agency and its adoption by the European Fighter Aircraft consortium have supported and encouraged the move towards object oriented methods. There are now at least two HOOD tool sets which are being sold in the open market and supported.

Like the other methods discussed, HOOD has diagrams with boxes and arrows. It also has the recursive principle, that is, a box can itself contain boxes and arrows: this is why 'hierarchical' is part of its title. But the HOOD diagrams are not data flow diagrams, the arrows indicate which objects use the services of other objects. In HOOD there are four different message types and two different object types to provide the synchronisation and parallel operation required in a high performance real time system. Such detail is not relevant to this paper, it is in Reference [9].

A Hood Object

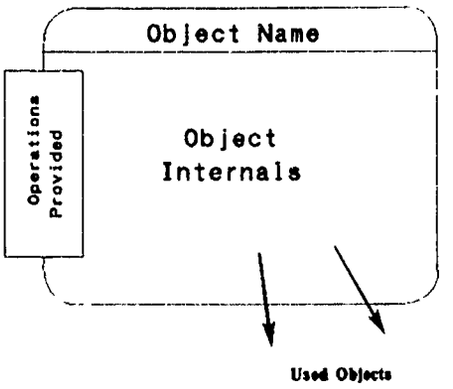


Figure 9

Restriction of communications between modules to requests for service is a powerful concept. Objects give a greater amount of encapsulation than the processor boxes in other methods; so, once its services have been defined, the design and implementation of an object can be carried out independently of the rest of the project. It retains and reinforces the gains made by earlier methods in data hiding and low coupling and the designer naturally tends to put related services in the same object. It therefore supports high cohesion and so adds to the gains in quality provided by earlier methods.

An additional advantage of object oriented methods is that they promote software reuse. The encapsulation provided by the concept of an object is so strong that it enables objects to be transferred

from one project to another unchanged if they provide appropriate services. If changes have to be made the change process is controllable, as it is restricted to changing the services provided by the object.

10. Formal Methods in Requirements and Design.

It has been known for more than ten years that the requirement for a computer system can, in principle, be encapsulated in a formal mathematical language. The requirement can then be proven complete and consistent. Further, the transformation to design and code can also be, in principle, formalised to ensure that the eventual code is proven to instantiate the requirement. This is most attractive. It suggests the possibility of proving that programs are correct. Testing does not do this, at best testing proves that the software is not incorrect in the aspect which is being tested. Yet, in practice, formal methods are rarely used. Many reasons have been advanced to explain this. From the point of view of this paper the relevant reason is that of scaling. Formal methods work well for small systems but the time and effort required to apply formal methods seems not to have a linear relationship with system size, it increases much more rapidly.

Current design methods give the designer great freedom in the way he chooses to decompose the system into activities and data flows so, if formal methods are to be applied, they have to be applied to the whole system as a single entity. At the current state of development of formal methods that is, in practice, impossible.

In an Object Oriented Design the objects have more solid boundaries and communication between them is more formalised than activities and the communication between them in current methods. This ameliorates the scaling problem. An object could be proven correct independently of the rest of the system. Then, since the inside of an object is hidden, a system or subsystem could be proven as a set of interacting objects.

11. Summary and Conclusion.

Software Quality is to a large extent determined by how well the developers follow the principle of Data Hiding, Low Coupling and High Cohesion. Current design tools provide good support for the first two of these concepts; support for high cohesion is less good. The newer methods, based on object oriented principles, retain the advantages gained by current methods and support high cohesion.

Formal methods, which to date have promised much more than they have produced, may gain a new lease of life as object oriented methods become more popular. The greater encapsulation provided by these methods may make the use of formal methods feasible on quite reasonably sized systems.

12 References.

1 P T Ward and S J Mellor, "Structured Development for Real Time Systems", Yourdon Press (1985)

2 "The Starts Guide", Second Edition, National Computer Centre (1981)

3 W.Stevens, G Myers and L.Constantine "Structured Design", IBM Systems Journal, Vol 13, No 2 (May 1974)

4 D L Parnas, "On the criteria to be used in decomposing software into modules", IEE Transactions on Software Engineering (December 1972)

5 "Special Issue on Mascot 3", Software Engineering Journal, Vol 1 No 3 (May 1986)

6 J R Cameron "JSP and JSD", IEEE Computer Society (1983)

7 M A Jackson "Constructive Methods of Program Design", Lecture Notes on Computer Science, Springer-Verlag (1976)

8 B Stroustrup "The C + + Programming Language" Addison Wesley (1987)

9 "HOOD Reference Manual", European Space Agency (1989)

Tool Supported Software Development
Experiences from the EFA Project

by
Werner M. Fraedrich, RDir, Dipl.-Phys.
Federal Ministry of Defense
RÜ P IV 3
D-5300 Bonn, West Germany

SUMMARY

EFA is a multinational project. As to pertinent data processing support, agreement had to be reached between the partner nations (both industry and government) The paper will show that, generally agreements were worked out by arriving at the lowest common denominator since none of the participating nations was prepared to accept standards established by another partner nation, which would have meant giving up its own standard.

The paper will address important additional information as well as experience gained to date:
* some general information on the EFA Project, including important determinations made
* the status of the software tool selection and procurement in the EFA Project
* a comparison between required and actual availability of software tools in the EFA Project

The paper will conclude by trying to point out what could be done in order to preclude the problems mentioned.

1. INTRODUCTORY REMARKS

EFA is a multinational project. As to pertinent data processing support, one could therefore not simply apply national standards and procedures. Rather, agreement had to be reached between the partner nations (both industry and government).

The following will show that important determinations have been made within the multinational framework at a very early stage. However, these were generally agreements worked out by arriving at the lowest common denominator since none of the participating nations was prepared to accept standards established by another partner nation which would have meant giving up its own standard. Later on, I am going to describe what comes of such a course of action.

Since these facts alone are not very meaningful, I will address important additional information as well as experience gained to date. To begin with, I am going to offer some general information on the EFA Project, including important determinations made. Following that, I will explain the status of the software tool selection and procurement in the project. After drawing a comparison between required and actual availability of software tools, I will conclude by trying to point out what could be done in order to preclude the problems mentioned.

2. REQUIREMENTS ESTABLISHED BY THE NATIONS

2.1 BACKGROUND

EFA is a quadrinational programme which is jointly carried out by the United Kingdom, Italy, Spain and Germany. Early on, France had also participated in the programme.

Due to the fact that the United Kingdom, Italy and Germany have been implementing the Tornado Programme together, the course of the EFA Project has already been set to a certain degree, which also shows in formulations of the earliest documents.

The requirements to be met by the development tools have been considerably influenced by Tornado. What had been agreed on trinationally, was now agreed by five nations or four, respectively.

2.2 EUROPEAN STAFF TARGET (EST, 11.10.84)

The EST formulations have been kept in fairly general terms and, actually require only
* in para 4.2.1 the use of a High Order Language, and
* in para 6.2.1 the use of development tools.

Para 4.2.1

"The common use of a High Order Language and a Bus-System is required."

Annex Q of the EST lists the following detail requirements:
* while France demands LTR
* the other nations call for Ada, and
* the United Kingdom also for Coral 66.

Para 6.2.1

"To build confidence in the ultimate quality of the hardware/software system the contractor is to submit a detailed proposal for a System Development Environment (SDE) for approval before installation. This should cover the following aspects:
* *codes of practice*
* *software tools*
* *test and integration*
* *use of language*
* *documentation standards."*

2.3 EUROPEAN STAFF REQUIREMENT (ESR, 09.12.85)

In the next phase (meanwhile without France), the formulations become considerably more precise:
* Ada is the preferred High Order Language
* In case of non-availability of an Ada Programming Support Environment (APSE), CORE/EPOS, a combination of tools, which had been agreed on in the case of Tornado, should be used.

Para 6.3.2.1

"Software is to be written in the HOL Ada throughout; exceptions may be made in particular areas ... all such exceptions are to be agreed with the Air Staffs."

Para 6.3.3

"To build confidence in the ultimate quality of the hardware and software system and the related documentation the contractor is to submit a detailed proposal for a System Development Environment (SDE) for approval by the nations. This is to cover the following aspects:

- ** standards and procedures*
- ** software tools and methods*
- ** use of language*
- ** ... ."*

Para 6.3.3.1

"In the absence of a full Ada Programming Support Environment (APSE) an EPOS/CORE system for all aspects related to:

- ** design and development*
- ** project management*
- ** configuration management*
- ** computer generated software documentation*
- ** ... ."*

2.4 EUROPEAN STAFF REQUIREMENT FOR DEVELOPMENT (ESR-D, 19.09.87)

Now, the formulations are becoming

* more precise on the one hand
  it is stated
  - when exceptions to using Ada are permitted, and
  - who has to approve the waiver (the Nations, rather than the Air Staffs)

* and more cautious on the other hand,
  as to the development tools, there is only in case of non-availability of an Ada Programming Support Environment a requirement that a tool combination be used like the one which had been agreed on in the case of Tornado (CORE/EPOS),

Para 6.3.2.1

"Software is to be written in an HOL. For all systems Ada shall be used, restrictions on the use of Ada or some of its features will have to be agreed between Industry and Nations."

Para 6.3.3

"To build confidence in the ultimate quality of the hardware and software system and the related documentation the contractor is to submit a detailed proposal for a System Development Environment (SDE) for approval by the nations. This is to cover the following aspects:

- ** standards and procedures*
- ** software tools and methods*
- ** use of language*
- ** ... ."*

Para 6.3.3.1

"In the absence of a full Ada Programming Support Environment (APSE) the SDE concept shall be based on the use of tools such as CORE/EPOS for all aspects related to:

- ** system requirement specifications/interface specifications*
- ** design and development*
- ** configuration management*
- ** computer generated software documentation."*

2.5 WEAPON SYSTEM DESIGN AND PERFORMANCE SPECIFICATION (WSDPS, 01.10.88)

The WSDPS - the development contract technical specification - does not directly specify the tools to be used. Rather, it says under "Software Design Principles"

Para 1.3.4.1

"Software shall be designed and developed using the SDE (see part III, para 2.6.6)"

and there, the following is stated:

Para 2.6.6

"The SDE shall provide a complete environment for system/software design and development.

It comprises:

- *a. CORE/EPOS and other tools agreed by the customer to be used for system/software design and documentation.*
- *b. Configuration Management and Modification procedures and tools... .*
- *c. EFA Software Standards ...*
- *d. Programme support environment for Ada and any other language ...*
- *e The tools/software to support software verification, testing, integration, validation and certification (including analysis tools).*
- *f. Generation of design documentation and cross reference between documentation levels."*

While industry has thus to propose the tools to be used, the Nations - particularly because of the influence on the in-service phase (Software Maintenance) - will have to accept that proposal.

2.6 PROJECT SPECIFICS

Before addressing data processing details regarding the EFA Project, let me mention a few project specifics.

The companies Eurofighter (EF) for the aircraft and EuroJet (EJ) for the engine are consortia formed by national companies. In the following, I restrict myself to the aircraft consortium, the Eurofighter Company, which is made up of the following national companies:

* Alenia (Italy)
* BAe (United Kingdom)
* CASA (Spain) and
* MBB/Dornier (Germany).

Each Eurofighter Partner Company (EPC) is within EF responsible (System Design Responsible Company = SDR Company), for specific tasks. Joint Teams consisting of personnel of all companies (in the case of avionic design such as the Avionic Joint Team (AvJT) based with BAe at Warton, which is also the leading office for all data processing matters) were formed which are responsible for the design of the various EFA systems.

There is an important project characteristic in that <u>all</u> documents (for example, also tender specifications) prepared by the SDR Company or the responsible team, have to be endorsed by the other companies, that is, having been prepared, they are forwarded to the partner companies for comments. Subsequently, Eurofighter Co. must approve and issue the documents.

This course of action is very time-consuming and it has happened that a report took nearly one year after preparation to arrive at the Nations and at NEFMA (our "NATO EFA Development, Production and Logistic Management Agency"). One reason for that procedure may well be that each EPC wants "to have a say" in each area - among other things, because of the national responsibility vis-à-vis its own government - and that the EPC is probably expected to do that on account of requirements established by the respective nations.

3. SELECTION AND PROCUREMENT OF SOFTWARE TOOLS

Work on EFA data processing aspects between the Nations, including NEFMA, and industry is done in the Systems Integration and Software Group (SISG) which held its first meeting from 17 to 19 September 1986, that is, more than two years before the development contract was signed.

3.1 PRECONDITIONS

Initially, basic matters had to be clarified and settled before the actual selection and procurement of tools could be tackled.

3.1.1 POLICY STATEMENT

The first activity engaged in by the data processing specialists of the four nations (from both, government and industry) was to determine data processing guidelines, according to which system/software design and development was to be accomplished. By the spring of 1987, a Policy Statement had been prepared and agreed, which, among other things, specified

* its applicability to specific software items (that is, all those required for the weapons system and the weapons system development).
* which significant management tools (plans, documents) were to be prepared (detailed plans in accordance with the EFA Software Standards), and
* on which standards these management tools were to be based (agreement was reached that system/software design and development be executed in accordance with DoD-STD-2167 thereby taking into account RTCA-DO 178/A).

3.1.2 EFA SOFTWARE STANDARDS

In line with the above determination and subsequent to the Policy Statement whereby the latter's requirements are taken into account, the EFA Software Standards were established beginning in spring 1987. They must be applied to all software including equipment software and by all equipment suppliers. Since 1989/1990, these standards have been available as binding Project Standards, and they also define

* the course of action to be taken by the software configurations management in the Eurofighter Software Configuration Management Plan (PL-J-019-E-1003) (SCMP, Issue 1 of March 1989)
* the development documents to be prepared, as well as the course of action to be taken in software development and certification, including software safety, in the Eurofighter Software Development Standard (PL-J-019-E-1006) (SDS, Issue 1 of March 1990))
* the use of tools agreed in the Eurofighter Software Methods and Tools Applications Standard (PL-J-019-E-1010) (SMTAS). This document will be prepared according to the project progress made. While the pertinent parts for System Design (Annex A) and Software Design (Annex B) have been available so far as Issue 2 of June 1990, we are still waiting for the first drafts for Software Coding (Annex C) and Software Testing (Annex D).
* the documentation of software development in the EFA Software Documentation Standard (PL-J-019-E-1011) (SDOS, Issue 2 of March 1990).This standard contains - in annexes - the model texts for the documents to be prepared according to the SDS. These model texts are prepared according to the project progress made. At this point in time, about 75 percent of them are available.

3.1.3 HIGH ORDER LANGUAGE

Right from the start the requirement was established by the participating Nations to use a High Order Language (HoL), and also which one - namely, Ada. In line with that determination, and beginning in 1987, that is, parallel with the EFA Software Standards, another Policy Statement was prepared by data processing specialists of the four nations (from both government and industry), which addresses the Ada compiler. With that Policy Statement, entitled "The use of Ada Compilers in the EFA Project" and dated 10 February 1988, among other things

* selection (procedure)
* development (course of action)
* validation matters, and
* use (updates, version control)

were determined.

It was an important task for the System Integration and Software Group to decide which compiler should be used. Prior to that, however, a decision had to be made as to which µ-processor should be used and which measures were to be taken in order to meet the reliability requirements for flight safety critical software.

In early 1988 Eurofighter Co. presented a study entitled "The Use of Ada for Safety Critical Applications".

This study demonstrated that Ada can be used for all safety critical applications. The reliability of Ada programs is comparable to that of assembler programs, if not greater, if

* any restriction on the use of the Ada language is strictly adhered to (Safe Ada), and
* the static code analysis at source code level (= Ada) is made.

Any reliability problems have been mainly seen in the compiler area. In order to preclude any possible compilation error as well as unforeseeable run-time behavior, a certain restriction on the use of the Ada language was agreed, together with initial target code verification.

3.1.4 STANDARD µ-PROCESSOR

In early 1988, Eurofighter Co. presented the results of a market survey in the form of a µ-Processor Report. In it, the Motorola MC68020/68881 was proposed to become the standard µ-processor for the EFA Project.

This µ-processor (and this is why we speak of a "standard µ-processor") will invariably be used by all subcontractors, that is, only one programming environment (compiler, run-time System, debugger, etc.) is required for the project.

Eurofighter and Motorola have yet to agree on a conditions, in which Motorola Co. would, among other things, guarantee

* supportability for a period of 25 years, beginning on the date the last aircraft enters into service, and
* enhanced radiation hardening.

In this connection, the main problem is that Motorola Co. is not prepared (and probably just cannot guarantee) that some years from now, the required production would still be based on the old technologies and old masks, when - in the meantime - production would have been modernized and the masks reduced in size.

3.2 THE EFA SYSTEM DESIGN ENVIRONMENT (SDE)

SDE includes tools for

* system design
* software design/development
* Ada compiler, and
* static and dynamic test tools

as well as

* IPSE with its standard tools.

3.2.1 TOOLS FOR SYSTEM DESIGN

The basic weapon system performance requirements had been known to industry already with the first phase documents, that is, very early. And the same documents also specified the tools to be used for weapons system design and development: CORE and EPOS. They could certainly have been used also in the early stages of the project in a meaningful way.

Whereas EPOS is mainly used

* to analyze and identify individual requirements from "plain english" texts
* to ensure requirement traceability
* to compile the Interface Requirement Documents

CORE is mainly used for the design process, i.e. to decompose high level requirements in a logical and consistent manner until a level is reached where the requirements are expressed in such sufficiently, precise detail to allow software design to begin.

Already on 21 October 1985, industry thus stated in a restrictive manner:
"It is anticipated that full facilities, both hardware and software, allowing direct entry from EPOS to CORE will be available to all partners not before August 1986."

The reason for this restriction was that the interface routines between the existing tools CORE (BAe Company) and EPOS (GPP Company), the so-called CORE-EPOS transformer, still had to be developed. The contract for that development was awarded in October 1988.

3.2.2 TOOLS FOR SOFTWARE DESIGN

As to software design and development tools, industry had conducted a market survey in 1987/88. An important task to be performed by that market survey was to determine the method to be used for software development. Following an analysis of the documents/proposals received, Eurofighter Co. recommended the Hierarchical Object Oriented Design, in short HOOD, to the Nations as method to be used for software development.

Here again, the market survey had shown that the tools required were not yet available in the configuration required by the project. Making use of the results of the market survey, a tender specification was prepared and - following evaluation of the proposals - in November 1989 the British company Integra Software (IPSYS) PLC (on behalf of Software Sciences) was awarded the contract to supply its "HOOD Toolset" as Hood tool.

Adaptation developments were also required in the following areas:

* traceability to CORE/EPOS and vice versa
* Ada code extraction
* security classification attribute for objects and its printout on documents and diagrams.

3.2.3 ADA-COMPILER

The Project Baselined Compiler to be employed according to the Policy Statement will be procured by both Eurofighter and Eurojet, and its use by the equipment suppliers - that is, for those who need it - has been ensured.

As to the Ada Compiler, industry conducted a market survey in 1987/88, whose results were incorporated into the tender specification. Subsequent to the request for proposal in early 1989 and their evaluation, the contract was awarded in February 1990 to SD-Scicon Company for XD-Ada

Adaptation developments were also required in the following areas:

* Target Run Time System for multi-µ-processor systems
* emulation of basic floating point operations (in case no mathematical co-processor is used)
* library of mathematical fixed and floating point functions.

3.2.4 STATIC AND DYNAMIC TEST TOOLS

As already shown, the reliability of Ada programs - where compiler errors are critical, rather than programming errors - is comparable to that of assembler programs, if not greater, provided that

* any restriction on the use of the Ada language is strictly adhered to (Safe Ada), and that
* the static code analysis at source code level (= Ada) is made.

As said before, target code verification will be applied as long as the compiler stability has not been proven.

Thus, when testing, appropriate tools must be used to check whether the programmer has observed the restrictions agreed.

While the SPARK Examiner (by PVL) is used to make the static code analysis, TESTBED (by Liverpool Data Research Limited (LDRA)) is used for the dynamic analysis. These tools, however, are only required to be used for "Risk Class 1 Software", that is, Flight Safety Critical Software.

The license agreements for these "off the shelf" tools were signed in mid 1990, following a market survey.

3.2.5 INTEGRATED PROGRAMMING SUPPORT ENVIRONMENT (IPSE)

The aim had been to find an Integrated Programming Support Environment, into which possibly all of these tools listed under paragraphs 3.2.1 through 3.2.4 could be integrated. In addition, it was to include standard tools, such as those for

* documentation
* configuration control.

As to the Integrated Programming Support Environment, industry had conducted an initial market survey at the end of 1987/in early 1988, to be followed later by the request for proposals. In the spring of 1990, Eurofighter Co. selected the tool "Perspective", offered by the British company System Designers. Since significant parts of the IPSE were not available as required, here again, a contract had to be awarded for adaptation developments also in the following areas:

* user interface (format of terminal windows)
* data management facilities
* database protection facilities
* configuration control
* integration of the interface control tool "Ingres".

4. AVAILABILITY VERSUS REQUIREMENTS

4.1 GENERAL

As can be seen from the observations, the entire system and software design and development of the EFA Project was to be carried out tool-supported and top-down. In this connection, a distinction is being made between tools for

* system design
* software design
* compiler, as well as for
* miscellaneous purposes, such as standards, test tools and IPSE.

Regarding all these tools, a decision has been taken in the meantime as to what is to be used by the aircraft consortium (EF) and its equipment suppliers, namely

* CORE/EPOS for system design
* the HOOD Toolset by IPSYS for software design
* SPARK Examiner by PVL for static code analyses
* TESTBED by LDRA for dynamic tests/Analyses
* XD-Ada by SD-Scicon as Ada compiler
* IPSE by System Designers based on Perspective.

4.2 THE TOOLS IN DETAIL

Let me now reverse the above sequence, that is, start with IPSE and finish with the CORE/EPOS system design tools.

4.2.1 IPSE

An initial version of IPSE should have been available in autumn 1990, but the acceptance tests for this version failed because of problems maintaining database integrity due to hardware problems. It was planned, that IPSE should be available in its entirety about mid 1991 but this will no longer be possible. Nevertheless, while this is not the optimum, it will not result in any significant problems or delays in the programme especially as the IPSE is only used by the EPCs.

4.2.2 STATIC AND DYNAMIC TEST TOOLS

Currently available versions of these test tools (SPARK Examiner and TESTBED) have been available since November 1990. This seems to be fully sufficient since the software tests - even those conducted by equipment suppliers - did not begin before late 1990. Updates of these commercially available tools are required.

4.2.3 THE ADA COMPILER

First versions of the Ada Compiler have been available since May 1990 for the standard µ-processor. In its entirety it will be available about 12 months later. This is fully sufficient since the equipment suppliers started coding not before the middle of 1990, and since the compiler versions available at that time had been sufficient for the initial work.

4.2.4 HOOD-TOOL

While first versions of the HOOD Tool have been available since December 1989, it will be available in its entirety in the middle of 1991. Here again, the tool has been available in good time for the development of both aircraft and equipment software.

4.2.5 CORE/EPOS

Even though the CORE and EPOS tools had already been released since 1987 for project use, their actual use had been limited since the CORE-EPOS transformer - just like the associated versions of CORE and EPOS - could not be used.

The development, which was started in October 1988, had taken much longer than was expected in the Eurofighter letter referred to earlier. It was not until the 16th meeting of the Systems Integration and Software Group held on 12/13 November 1990 that industry reported *"now the transformer can be successfully used on the project"*.

But due to the necessary paperwork its release for use on the project only took place in early December 1990.

4.2.5.1 REASONS FOR LATE AVAILABILITY

The first quadrilateral document, the ESR, was signed in late 1985. This was the beginning of the joint definition phase. However, some time passed - although the participating nations and industry had reached early agreement on the

tools - until CORE/EPOS, and even parts of them, had been available to all development teams of the aircraft companies, to say nothing of the sub-contractors. Apart from the implementation of desired/required - but as yet not released - requirements, there were also general commercial issues that hindered the rapid use of the tools by all participants, such as

* individual or project license
* which version
* how many "systems" per company
* how many users per "system".

Regarding the EFA project, industry has thus worked for a long time either without the tools or with the tools, but only to a very limited extent.

4.2.5.2 THE CONSEQUENCES OF LATE AVAILABILITY

According to the basic documents, such as EST, ESR, ESR-D and WSDPS, the whole system design should have been carried out tool-supported and top-down; that is, all concept and definition work should have already been accomplished with data processing support. This, however, was not the case.

The technical requirements to be met by the weapon system were laid down in the Weapon System Design and Performance Specification; naturally, in "plain language" since this was an annex to the main development contract.

A conversion to EPOS was not carried out until one year later (1989). Thus, the main requirements to be met by the weapon system had not yet been specified for the first Functional Requirement Documents (FRDs), such as Avionic System FRD and Subsystem FRDs, in a Requirement Data Base. It was not until preparation of later versions of these FRDs, that the Requirement Data Base could be assessed.

Whether or not system development could have been carried out that rapidly and smoothly given early availability of the transformer, so that one really could speak of a top-down design, cannot be determined by this briefer. In this connection, the top-down design means to make the following approach:

* At first, preparation of the System FRDs for the avionic, flight control and utility control systems on the basis of the Weapon System Design and Performance Specification (1st level)
* subsequently, preparation of the Subsystem FRDs (2nd level)
* based on that, preparation of the LRI Processing/Software Requirement Specifications and of the equipment specifications (3rd level).

Above all, the determination as to which requirement/task is to be met or carried out by hardware, and which one by software, will be decided not before the 2nd system design level.

I consider the late availability of all parts of the system design tools CORE/EPOS (that is, including the transformer, the "problem child" of that tool combination) as the main reason that departure from this ideal top-down design.

Parallel to the preparation of the design documents (Avionic System FRD and Subsystem FRDs), that is, before system design of the 2nd level (Subsystem FRDs) had been completed, the equipment specifications were established. Otherwise - as stated by Eurofighter Co. - the fixed target dates of the development programme - particularly the first flight of the first prototype (P01) and/or that of the first avionic prototype (P05) - could not be met.

At an Design Review, held in January 1989, the Avionic Joint Team explained in detail how the basic documents (ESR, WSDPS) and the design documents (FRDs, LRI Processing Specifications) should serve to carry out system design and development of both hardware and software by means of

* a market survey (based on ESR)
* preliminary equipment design requirements (based on logistic requirements of the WSDPS and "initial unit functions" of the System FRDs)
* tender specifications (among other things based on the detailed functions of the Subsystem FRDs)
* contract specifications.

The customer (NEFMA/Nations) is not enthusiastic about this course of action since it represents *"rather an equipment specification loop than an equipment specification route"*, and since the proposals may have a strong impact on the system/subsystem design.

5. "LESSONS LEARNED"

When considering all tools, one thing becomes very clear: the design tool is most critical. All other tools are needed later on in the project life cycle; that is, there is usually sufficient time for their

* selection
* procurement, and
* any adaptation developments, if required.

Especially in the case of system design tools, their absolutely necessary use may be easily put at risk as important development steps are initially taken without them, thus making early decisions, which could be corrected, not at all or only with great difficulty (and usually turning out to be also very costly). To make it clear, use of tools will not prevent design errors. But if the designer is using such tools, his attention is in many cases called early enough to inconsistencies and other deficiencies in his design that correction will be possible without too great difficulties (and usually also not great cost).

The experiences gained from the EFA Project to date, clearly show that

* international (NATO) standards for system- and software development, including documentation and tools, as well as
* early decisions on the design tools to be used (that is, already at the start of multinational cooperation; however, not only between the nations, but also between the participating industrial companies)

are urgently required.

Moreover, a top-down design as described earlier would certainly be desirable. This, however, may be possible only if - at the time of signing the main development contract

* the market survey has been completed

* the system design has been completed, if possible down to the LRI Processing Specification level, but at least to the Subsystem FRD level, and
* the tender specifications have been prepared for all equipments/subsystems/systems.

This, however, will not be realizable since the basis of system design - that is, the technical specification (in our case, the WSDPS) - is usually agreed in a binding manner together with the main development contract.

This dilemma could probably be evaded only if development is carried out in stages, that is
* at first, system design until the above preconditions have been met, and
* then development with the invitation to tender for the equipments, is not started before the second stage.

However, I cannot really say whether such a course of action would actually be possible and, if so, whether it would also help to resolve the problems described.

Military and Civil Software Standards and Guidelines for Guidance and Control

by
K.W. Wright
Smiths Industries Aerospace and Defence Systems
Bishops Cleeve
Cheltenham
Glos
U.K.

SUMMARY

The two most widely used standards covering the development of software in the military and civil avionics industries are DOD-STD-2167A and RTCA DO.178A/EUROCAE ED-12A respectively. This latter document is currently undergoing extensive update by RTCA Special Committee 167 and EUROCAE Working Group 12, with a planned document re-issue date of the end of 1991. This paper compares DOD-STD-2167A with the work currently being undertaken by SC.167/WG.12.

1. INTRODUCTION

As a direct consequence of the size of the US defence market, the most commonly used standard covering the development of military guidance and control software is DOD-STD-2167A, 'Military Standard, Defence System Software Development', (Ref. 1).

RTCA DO-178A/EUROCAE ED-12A, 'Software Considerations in Airborne Systems and Equipment Certification', (Ref. 9), has been used by the world's civil aviation certification authorities as the basis for clearing avionic equipment and systems containing software, since 1985. It should be noted that software is only certificated as an integral part of equipment or a system, never stand alone. The document is currently undergoing an extensive revision by RTCA Special Committee 167 and EUROCAE Working Group 12. The current target date for the publication of DO-178B/ED-12B is December 1991.

This paper compares the requirements contained in DOD-STD-2167A with the discussions taking place within SC-167/WG 12. It must be emphasised that the revised DO-178A/ED-12A guidelines, discussed in the text, are based on the author's understanding of the status of the activities within SC-167/WG 12 as at the end of January 1991.

Note: The terms SC-167 and DO-178B are used subsequently in the text to reflect the activities jointly being undertaken by SC-167 and WG 12, and the current working position of the revision to DO-178A/ED-12A, respectively.

UK DEF STAN 00-55, 'Requirements for the Procurement of Safety Critical Software in Defence Equipment', was issued in May 1989 as a draft interim standard. A number of its requirements, including mandatory use of formal mathematical methods, and use of an organisationally independent verification and validation team, gave rise to strong reaction from UK industry. As a consequence of the large number of comments received, the document has been undergoing, what is believed to be, significant modification. It had been intended that this paper would compare the contents of the revised standard with the requirements of DOD-STD-2167A and DO-178B. However, at the time of writing, the revised document has not been released, the comparison has therefore not been possible.

2. HISTORY

DOD-STD-2167

DOD-STD-2167 was initially released in June 1985 with the aim of reducing the occurrence of programme cost and schedule overruns and at improving the quality and maintainability of software products.

Revision A of the standard was released in February 1988 in order to take account of the comments and criticisms received following the practical application of the requirements specified in the original document. In particular, the new issue addresses the identified deficiencies with respect to incompatibilities with the use of Ada and with new and evolving software engineering technologies.

RTCA DO-178

RTCA DO-178 was first published in January 1982 following agreement, between the certification authorities and industry, on the need for guidelines covering the certification of avionic equipment containing digital computers. The document was specifically aimed at providing guidance on how the authorities' requirements, particularly with respect to safety, might be satisfied.

In 1983 it was decided that the document should be revised to reflect the experience gained during the initial period of field application. Revision A of the document was published in March 1985.

Following the circulation of a questionnaire by the FAA to industry, requesting information on the experience gained and problems identified with using DO-178A, RTCA established SC-167 in October 1989. SC-167 was tasked updating the document with the aim of making software production more effective and efficient, and at the same time, enable a more consistent interpretation.

A copy of the Terms of Reference for SC-167 are included as Appendix 1.

3. PURPOSE AND SCOPE

DOD-STD-2167A

The main purpose of the document is to provide a procurement specification for deliverable software in the form of Computer Software Configuration Items (CSCI's). However, the requirements may also be applied to the development of software delivered as part of Hardware Configuration Items, firmware and non-deliverable software.

The document defines the software development process as shown in Figure 1. The fact that the process begins and ends with system related activities, emphasises the importance of the need to control the interface between the system and software development activities.

DOD-STD-2167A, in addition to the specific requirements it contains, calls up five further standards covering configuration control, specification practices, management and technical reviews and audits.

RTCA DO-178B

The purpose of the document is to identify and describe software development and management methods and techniques, whose application will result in software products which perform their intended function in compliance with airworthiness safety requirements.

The guidelines are intended for use in either the development, modification or approval of systems and equipment. They may also be used for the qualification of software tools and methodologies, used in the certification process. The guidelines are not intended to be applied to user selectable databases or to support software which is not safety related.

The document only provides guidance on software considerations, it is not within the scope of DO-178B to provide guidance on systems processes. The document only refers to items of information which are expected to be transmitted between the system and software processes.

4. MANAGEMENT OF THE SOFTWARE DEVELOPMENT PROCESS

DOD-STD-2167A

The equipment supplier is required to establish a project software development life-cycle consisting of the activities identified in Figure 1. The activities are permitted to overlap and be performed iteratively or re-cursively.

Details of how the identified activities are to be performed and how they relate to the formal reviews and audits, required by the contract, are documented in the project Software Development Plan (SDP). With the exception of scheduling data, all updates to the SDP require customer approval. Any specific aspects of the proposed development programme which are considered to be a potential source of technical, cost or schedule risk, must be identified, analyzed, prioritised and monitored.

The potential need for subcontractor management, establishing an interface to an Independent Verification and Validation body and security are also identified.

The supplier is required to monitor the utilisation of processing resources throughout the programme and to re-allocate resources as required in order to meet the reserve requirements.

If use of a High Order Language is not mandated in the contract, the choice of language to be used is subject to approval by the customer.

The need to plan the transfer of the product from a development to a support environment is identified together with the associated requirements for maintenance and documentation specific to the support activity.

RTCA DO-178B

The primary purpose of the guidance material which will be provided in relation to the management process, is aimed at providing the certification authorities with confidence that the software products meet their requirements with respect to safety.

The software development process model likely to be adopted by the document will be based on a subset of the proposals made by the IEEE, (Ref 2). This defines the software lifecycle as a sequence of processes, see Figure 2. Each process includes a number of activities which must be completed in order to complete the lifecycle, however, activities need not be completed in a single 'pass' through the process.

The appropriate software lifecycle is determined for each software development task. Large developments could be broken down into separate tasks, each having a distinct lifecycle.

A software lifecycle is made up of software development and integral processes. Development processes are product orientated activities, i.e. planning, requirements analysis, design, code and integration. Integral processes provide engineering support and assurance functions such as verification, configuration management, quality assurance, etc. These latter processes are required to ensure the completion of the lifecycle and the quality of the end product.

The planning process is fundamental to the development of good quality software, it is during this phase that the specific lifecycle(s) for the

project is (are) defined. This planning activity only relates to software development and is subset of overall project management.

Note: Business or commercially related programme management activities are currently being addressed by the ARINC/AEEC Software Management Sub-Committee which is developing Project Paper 652 'Guidance for Avionics Software Management'.

5. SYSTEM DEFINITION

DOD-STD-2167A

The standard divides the System Definition phase into two separate activities. System Requirements Analysis, which involves a review of the (customer) system specification to remove any deficiencies eg ambiguities, omissions, inconsistencies, etc., and System Design, where the requirements are allocated between hardware, software and the 'user'. This latter activity results in the system being partitioned in to hardware and software configuration items, and, where appropriate, manual operations.

In order to minimise the chances of the system entering a hazardous state during operation, the supplier is required to perform a safety analysis. Any potentially hazardous events must be clearly identified and documented.

As part of this phase the supplier is required to develop preliminary software and interface requirements for each CSCI.

The output of the System Definition process provides a 'functional baseline' on which the software requirements will be based.

RTCA DO-178B

As stated in section 3, systems related activities are outside the scope of DO-178B, SC-167 is only addressing those items of information which pass between the systems and software processes.

Note: SAE, at the request of the FAA, has established a group tasked with developing system integration requirements. This group is working closely with SC-167 to establish an interface between the two activities.

The purpose of the system/software interface is to identify the input requirements, necessary to enable the software development process to proceed, and the outputs of the process, needed for use in system validation. One of the major issues being addressed is the traceability and accountability to those system requirements which are related to system safety. The aim being to establish safety directed software development process. Figure 3 contains an overview of the flow of information between system safety related processes and the currently proposed software development processes.

System safety requirements may be satisfied in two ways, namely, preclude by design or prove absence through verification. All safety requirements allocated to software will be documented in the software requirements. The requirements will also specify the software criticality level.

SC-167 has currently identified five levels of software criticality, A through E. The levels are tied to the failure condition categories as defined in AMJ 25-1309 i.e. Catastrophic, Severe/Major (Hazardous), Major and Minor, plus a category corresponding to the case where there are no safety implications. The selection of software level will be based on the potential contribution of software to a failure condition, as determined by a system safety assessment activity. Appendix 2 contains definitions of the failure condition categories and the corresponding software levels.

6. SOFTWARE REQUIREMENTS ANALYSIS

DOD-STD-2167A

The purpose of this phase is identified as an analysis of the system requirements allocated to each CSCI and the definition of a complete set of software and interface requirements. Included in the analysis are the processing resource and reserve requirements for each configuration item including throughput, memory and I/O port loading.

The resultant CSCI software and interface requirements form the 'allocated baseline' for the software design process.

RTCA DO-178B

The inputs to this activity are identified as system requirements allocated to software, specific safety related requirements, software criticality level, project standards and the approved project approaches to software development, quality assurance and configuration management. Subsequent to the initial pass through the process, inputs will take the form of requirement changes and feedback from later processes.

The activities associated with this process are aimed at generating a well defined and traceable set of software requirements for use by subsequent processes. Any deficiencies identified during this process must be fed back into the system process.

The only impact of software criticality level on this process is likely to be in the level of detail provided by the documentation.

7. SOFTWARE DESIGN

DOD-STD-2167A

The standard separates this activity into Preliminary and Detailed Design. The process involves the partitioning of each CSCI into Computer Software Components (CSC's) and Computer Software Units (CSU's). An example of a 'typical' decomposition of a CSCI is shown in Figure 4.

The preliminary design activity results in the definition of the high level design of each CSCI and allocates software and interface requirements to CSC's. This activity also involves the preliminary design of the interfaces external to the CSCI.

The detailed design activity involves the breakdown of the high level design to unit level, by allocating software requirements from the CSC's to the CSU's. The detailed design of the interfaces external to the CSCI is also developed during this phase.

There is a requirement to consider the use of Non-Developmental Software (NDS) ie re-usable software, commercial off-the-shelf (COTS) software, and Government furnished software, during the software design process. Any use of NDS software must be agreed by the customer, identified in the project plans and documented in accordance with the requirements of the standard.

In order to assist the future understanding of both the high level and detailed design, supporting information, such as rationale, results of analyses and trade-off studies, etc., must be retained and documented.

RTCA DO-178B

The design process is divided into two parts, high-level architecture definition and detailed software design. The architecture definition activity involves the allocation of requirements to high level software functions, plus the definition and design of the hardware/software and software/software interfaces. The detailed design process includes definition of the low-level structure of the software tasks and the allocation of requirements to software units.

The inputs to the process, in addition to the software and interface requirements, include design standards and the approved project approach to tools, verification, configuration management and quality assurance. Following the initial pass, inputs also arise as a result of requirement changes and feedback from subsequent processes.

The document will also address the use of software developed as part of another project i.e. re-used software, and software supplied by a third party e.g. Ada run-time libraries and COTS software. Consideration will be given to the consequential impact on the design process of including software not necessarily developed to the same standards and procedures.

Guidelines are also being developed to cover special design requirements for developing 'User Modifiable' software. This is defined as software which may be modified, within fixed constraints, by the end user, without any requirement for re-certification. The limits, within which, the user is permitted to change the software, are identified and approved by the certification authorities, at the time the equipment is cleared to enter service.

The process product is a design description, including traceability back to the software requirements. The only impact of software criticality level on this process is likely to be in the level of detail provided by the documentation generated.

8. SOFTWARE IMPLEMENTATION

DOD-STD-2167A

The standard specifies language independent coding standards and the supplier is responsible for developing project specific codes of practice in accordance with these requirements. Following acceptance by the customer, the standards are employed in the implementation of the requirements of each CSU.

RTCA DO-178B

The coding process involves the production of source code based on the software design and requirements. It will be generated in accordance with project coding standards and subject to the approved quality assurance and configuration management procedures.

Any deficiencies identified in the requirements handed down must be fed back to the preceding processes for correction or clarification.

9. SOFTWARE VERIFICATION

DOD-STD-2167A

The verification requirements contained within the standard are distributed over a number of activities and involve reviews, product evaluations, development testing and formal qualification testing. An important aspect of the verification process is the need to provide documented traceability of the system requirements allocated to each CSCI, its CSC's and CSU's and to formal test cases.

The supplier is required to conduct or support formal reviews at various stages of the project as indicated in Figure 1. All technical reviews are required to be performed in accordance with MIL-STD-1521 (Ref. 8). The standard does allow sufficient flexibility to enable reviews to be planned and scheduled to meet project needs, multiple reviews (e.g. PDR's and CDR's) are also permitted.

Evaluations are required to be performed on all deliverable software and documents at specified stages in the development life-cycle. In order to ensure adequate objectivity, personnel involved in the development of a product, may not conduct its evaluation, but, members of the engineering team are permitted to participate in the activity. All records associated with evaluations, including problem identification and corrective action, must be retained and available for review by the customer. Details of default evaluation criteria are provided by the standard, but the supplier is free to propose alternate or additional criteria, subject to customer approval.

The supplier is required to perform development testing on all CSU's and integration testing on all CSC's to ensure that they satisfy their specifications. In both cases the test procedures and results must be recorded and retained.

Following completion of the integration testing activity, the process by which the CSU's and CSC's are combined to form an integrated software product i.e. a CSCI, the product together with its associated formal test documentation, is reviewed to determine its readiness for Formal Qualification Testing (FQT).

The supplier is required to develop and document plans and procedures for performing FQT in a Software Test Plan (STP). Following its acceptance by the customer, with the exception of scheduling information, all changes to the STP must be subject to customer approval.

The STP is required to contain details of the software development environment to be used, including information relating to its verification, configuration management and maintenance. Further, in order to ensure the required degree of objectivity, persons involved in the development of the software, may not conduct the FQT, however, members of the engineering team may participate in the activity. The STP will contain details of how the stipulated level of independence is to be achieved.

FQT activities related to each step in the software development life cycle, shown in Figure 1, are identified within the standard.

When the CSCI testing activity has been completed, the software product must be integrated into the system and the system validated against the requirements identified and documented during the system requirement analysis phase.

Functional and Physical Configuration Audits must be performed on each CSCI on completion of either the CSCI, or system integration activities. The product of this activity provides the 'product baseline' for all subsequent activities. On completion, all documentation associated with the FQT and audit activities is required to be reviewed prior to delivery to the customer.

RTCA DO-178

The document will identify verification as an integral process and as such, verification related activities are carried out as part of, or in parallel with, the development processes. The guidelines will divide the verification process into two principle activities ie reviews and analyses, and testing. Emphasis will be on the importance of reviews during the requirements analysis, design and coding processes and on testing during the integration process. The need to perform analyses during all phases of the life-cycle will also be emphasised.

Reviews and analyses are required to ensure the correct and complete translation of, system requirements to software requirements, software requirements to design and eventually to code. At each stage, any requirements related to safety must be identified and addressed specifically. The review activity must include checks to assure adherence to the appropriate project standards, whilst the analyses verify compliance with, and traceability, to higher level requirements.

The guidelines will identify the fact that since the design process make take the form of a number of iterative steps the verification activity itself may also be iterative.

The results of the verification activities must be recorded and retained, details of problems identified, logged and corrective action tracked. Traceability is also required from system requirements to the verification products. In some cases, particularly where the software is classified as safety critical, a review of the verification products may be required, in order to provide assurance of the completeness i.e. comprehensiveness, of each verification activity.

Testing is identified as providing evidence that a function has been implemented correctly, and as a means of identifying interface definition deficiencies. The benefits of addressing the specific needs of the testing activity during the requirements analysis and design phases will be identified, as will the need to test for both normal and error conditions.

The document will identify three categories of testing ie low-level, software integration and hardware/software integration. It will also emphasise that the majority of the test procedures should be developed from the software requirements. There will be a requirement to carry out a structural coverage analysis on the requirement based test procedures and source code, to demonstrate that the

code structure has been adequately exercised. This latter activity may identify the need for additional test cases.

The guidelines will identify that the depth and scope of the verification process is governed by the criticality level of the software. The document will provide guidelines on the minimum verification requirements for each software criticality level.

In order to ensure that the right level of objectivity in the various verification activities, the verification of the individual products of the software development process, must be performed by someone other than the person(s) who developed the product.

10. SOFTWARE CONFIGURATION MANAGEMENT

DO-STD-2167A

The supplier is required to establish and maintain a software development library together with the necessary supporting plans and procedures. The latter are required to enable the software and related documentation produced during the development process, to be identified and controlled. All problems identified with software and/or documentation placed under configuration control, or with the development life-cycle processes, are required to be subject to a corrective action procedure. This process will include an analysis to detect any adverse trends in the problems identified.

Minimum requirements with respect to configuration identification, control and status accounting are provided, together with category and priority classifications for problem reporting. The standard also calls up DOD-STD's 480 and 481 (Refs. 6 and 7 respectively).

The specific configuration management requirements related to each step of the software development life cycle, shown in Figure 1, are identified within the standard.

RTCA DO-178B

To date, SC-167 has not reached any firm conclusions with respect to guidelines for software configuration management.

It has been agreed that there is a need for a disciplined configuration management approach throughout the software product life-cycle. It is considered important that identifiabie configuration items and baselines are established to enable the software to be controlled, documented, verified, maintained, reviewed and audited.

It is also accepted that there is a requirement for a formal change control procedure to be in place which includes a method of recording problems and tracking their resolution.

11. SOFTWARE QUALITY ASSURANCE

DOD-STD-2167A

The standard does not address the requirements for quality assurance explicitly, however, the need for independence in carrying out the evaluation and testing activities can be classified as a quality assurance requirement. Also, a number of the default product evaluation criteria, identified in the document, can be considered to be quality assurance related.

Military project software quality assurance requirements are provided by other documents such as DOD-STD-2168 and AQAP 13 (Refs. 4 and 5 respectively).

RTCA DO-178B

SC-167 has yet to reach any firm conclusions on guidelines for software quality assurance.

The objectives of software quality assurance process have been identified as:

(a) to assure that plans, standards and procedures are established and 'fit for purpose',

(b) to assure that approved plans, standards and procedures are being complied with,

(c) to assure that the evidence eg records, etc., provide confidence that the software products conform to the established technical, procedural and safety requirements,

(d) to ensure that the entire development life-cycle process is reviewed and any deficiencies identified and corrected.

The above will be achieved by closely monitoring and auditing the project activities, whilst maintaining the independence of the quality assurance function.

12. DOCUMENTATION

DOD-STD-2167A

The content and format of each deliverable document is defined in a Data Item Description, (DID), which is considered to form part of the standard. A list of the deliverable documents is provided in Figure 5.

The standard requires more documentation to be made available for review or audit than is identified as deliverable. The project specific deliverable documentation requirements are provided in the Contract Data Requirement List (CDRL). The SDP must include the justification for not producing non-deliverable documentation identified in the standard. The format and content of all non-deliverable documentation to be generated must also be specified in the SPD.

Each document DID provides very detailed information on layout, sub-paragraph content, the applicable sections of relevant standards, and at which review(s) the document should be presented for approval.

In order to minimise unnecessary overheads MIL-HDBK-287, (Ref. 3), has been produced to assist projects to 'tailor' the documentation generated to their specific needs. Tailoring may be used to eliminate either non-applicable sections of individual documents or complete documents. Project specific instructions for tailoring DOD-STD-2167A requirements will normally be specified in the Statement of Work (SOW), whilst tailoring instructions for the DID's will be specified in the CDRL. The extent to which project specific tailoring has occurred must be identified in the SDP.

RTCA DO-178B

The document will identify the information which will be required to support system/equipment certification. The information needed can be categorised as follows:

(a) Process Definitions
These will take the form of plans and standards which will detail the strategies to be followed and the methods and tools to be employed. Information relating to the configuration of the support/development environment will also be required.

(b) Process Outputs
These may take the form of requirements and design documentation, source code, and verification procedures and results. The information supplied will provide the evidence required to prove that an activity has been completed satisfactorily, in compliance with its plans and standards. Also, to enable the software products to be controlled and maintained, configuration Index's must be produced.

(c) Summary Information
Both the certification plan and the accomplishment summary are used to optimise the certification process.

DO-178B will only provide guidelines on the information to be supplied. Apart from grouping the information under headings eg Software Quality Assurance Plan, Software Requirements, etc., no specific requirements with respect to format and structure will be provided. The information may be made available in a number of forms such as individual documents, combined into larger documents, distributed across several documents, or on magnetic media. The only requirements are that it must be available in form which can be reviewed efficiently, and that the mechanism chosen must be identified in the Certification Plan.

13. CERTIFICATION

DOD-STD-2167A

The standard has been developed to establish a common interface between customers, suppliers and maintainers. As such, it is not intended to be used directly, to provide a third party, such as the civil aviation certification authorities, with the level of visibility they require. The document therefore does not contain any specific requirements related to this activity.

RTCA DO-178B

The primary purpose of the guidelines is to enable the supplier to provide the certification authorities with proof that the software content of the system or equipment has been developed in a structured manner. This proof may involve an audit of the development process employed, a review of the project documentation, concurrence with the suppliers statement of compliance, or some combination of all three.

The level of involvement by the certification authorities will be dependent on the system safety assessment and the resultant criticality level given to the software functions. The level of rigor to be applied, particularly in relation to the verification process, and the amount of data required as deliverables, will be dependent on the potential impact of any software errors on the safety of the aircraft.

Based on knowledge of the software level(s), the supplier is required to develop plans covering certification, quality assurance, configuration management and verification. These plans will be used to inform the certification authorities of the methods, tools and techniques which will be used to design, implement, verify and control the software development process. The plans should be prepared in advance of the software development life-cycle activities.

As a minimum, the certification authorities will require delivery of the following plans. Software Aspects of Certification, Software Quality Assurance and Software Configuration Management, together with a Software Accomplishment Summary. Additional documentation deliveries will depend on the criticality of the software. The supplier will be required to propose a set of deliverables as part of the Certification Plan.

Any documentation submitted as evidence of compliance must be that which controls, or results from, the software development process. With the exception of the Accomplishment Summary, no document should be produced solely for use by the authorities.

If a supplier wishes to reduce or eliminate particular verification activities by using a software tool, the

certification authorities will require the tool to be 'qualified'. Guidelines will be provided for determining if software tool qualification should be sought and, if so, the process to be followed in order to obtain certification authority approval.

14. CONCLUSIONS

DOD-STD-2167A was developed principally as a procurement standard and as such, it provides detailed requirements with respect to the software development documentation required as deliverables. It also identifies deliverable documents which are specific to the needs of users and software support personnel. Although there is a requirement to carry out a safety analysis to identify any safety related risks, no specific hazard classification related variation in requirements is identified. Variation may be possible by means of the tailoring information contained in MIL-HDBK-287.

The guidelines provided by RTCA DO-178B are primarily aimed at giving the certification authorities the assurance that the software has been developed in accordance with the regulations, particularly those related to safety. A great deal of emphasis will therefore be placed on the verification, assurance and control related activities. Information will also be provided on how the requirements may be modified for the different software criticality levels.

It should be emphasised that the civil certification authorities do not certificate software stand-alone, software will only be certificated as an integral part of equipment or a system. DOD-STD-2167A does cover the situation where the procured item is a software product ie a CSCI.

Provided the additional documents required by RTCA DO-178B are available eg Accomplishment Summary, Quality Assurance Plan, etc., and the supplier can demonstrate that the contents of the documents, produced in accordance with DOD-STD-2167A, comply with the RTCA DO-178B guidelines, then it should be possible to obtain certification authority approval for a CSCI as part of a system or piece of equipment.

However, due to the degree of document format and content flexibility likely to be available within the guidelines of RTCA DO-178B, the probability of it being acceptable for a procurement against the requirements of DOD-STD-2167A is not high.

As stated previously the comments on the likely contents of DO-178B are based on the author's understanding of the status of the discussion at the end of January 1991. The content and structure of the document may change significantly by the time it is finally issued.

REFERENCES

1. DOD-STD-2167A 'Defence Systems Software Development' 29 February 1988

2. Standard for Software Life Cycle Processes (Preliminary) IEEE, P1074/DS, 1 December 1989.

3. MIL-HDBK-287 A Tailoring Guide for DOD-STD-2167A,'Defense System Software Development', 11 August 1989.

4. DOD-STD-2168 'Defense System Software Quality Program',29 April 1988.

5. AQAP 13 NATO Software Quality Control Systems Requirements, August 1981

6. DOD-STD-480A Configuration Control - Engineering Changes, Deviations and Waivers, 12 April 1978

7. DOD-STD-481 Configuration Control - Engineering Changes, Deviations and Waivers (Short From)

8. MIL-STD-1521 Technical Reviews and Audits for Systems, Equipments and Computer Software.

9. RTCA DO-178A/EUROCAE ED-12A, Software Considerations in Airborne Systems and Equipment Certification, October 1985

APPENDIX 1
TERMS OF REFERENCE
Special Committee 167
DIGITAL AVIONICS SOFTWARE

Special Committee 167 shall review and revise, as necessary, RTCA Document DO-178A, "Software Considerations in Airborne Systems and Equipment Certification".

GUIDANCE:

In accomplishing its work the Special Committee should recognise the dynamic, evolving environment for software requirements, software design, generation, testing and documentation, and formulate a revised document that can accommodate this environment while recommending suitably rigorous techniques. To accomplish this revision, the Special Committee should consider the experience gained through the field application of the guidance material contained in DO-178, and DO-178A, as well as the results of recent research in software engineering. SC167 should also recognise the international implications of this document and, therefore, should establish a close working relationship with EUROCAE (which has become the normal practice in RTCA Committees). An objective should be to achieve a common/parallel RTCA/EUROCAE document. The Special Committee should focus this review to address the following areas:

1. Examine existing industry and government standards and consider for possible adaptation or reference, where relevant.

2. Assess the adequacy of existing software levels and the associated nature and degree of analysis, verification, test and assurance activities. The revised process criteria should be structured to support objective compliance demonstration.

3. Examine the criteria for tools to be used for certification credit (e.g. development, configuration management and verification tools).

4. Examine the certification criteria for reusable software, off-the-shelf software, databases, and user-modifiable software for the system to be certified.

5. Examine the certification criteria for architectural and methodological approaches used to reduce the software level or to provide verification coverage (e.g. partitioning and dissimilar software).

6. Examine configuration control guidelines, quality assurance guidelines, and identification conventions, and their compatibility with existing regulatory requirements for type certification, in-service modifications, and equipment approval.

7. Consider the impact of new technology such as modular architecture, data loading, packaging and memory technology.

8. Examine the need, content, and delivery requirements of all documents, with special emphasis on the accomplishment summary.

9. Define and consider the interfaces between the software and systems development life cycles.

10. Review the criteria associated with making pre- and post-certification changes to a system.

11. Consider the impact of evolutionary development and other alternative life cycles to the model implied by DO-178A.

APPENDIX 2
RTCA DO-178B (DRAFT)

SYSTEM CRITICALITY CATEGORY AND SOFTWARE LEVEL FAILURE CONDITION CATEGORIES

The failure condition categories described below are those accepted by the aviation community and the certification authorities for use in equipment and system certification. The categories are based upon the severity of the effects of failures or design errors on the aircraft, crew, and occupants. The categories are:

a. Catastrophic - Failure conditions which would prevent continued safe flight and landing.

b. Hazardous/Severe-Major - Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be:

* a large reduction in safety margins or functional capabilities.

* physical stress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely; or

* serious or fatal injury to a relatively small number of the occupants.

c. Major - Failure conditions which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or some discomfort to occupants.

d. Minor - Failure conditions which would not significantly reduce aircraft safety, and which involve crew actions that are well within their capabilities. Minor failure conditions may include, for example, a slight increase in crew workload, such as routine flight plan changes, or some inconvenience to occupants.

e. No Effect - Failure conditions which do not effect the operational capability of the aircraft or increase pilot workload.

SOFTWARE LEVELS

a. Level A - Software who anomalous behaviour, as shown by a system safety assessment, would lead to a failure of system function resulting in a catastrophic failure condition for the aircraft.

b. Level B - Software whose anomalous behaviour, as shown by a system safety assessment, would lead to a failure of system function resulting in a hazardous/severe-major failure condition for the aircraft.

c. Level C - Software whose anomalous behaviour, as shown by a system safety assessment, would lead to a failure system function resulting in a major failure condition for the aircraft.

d. Level D - Software whose anomalous behaviour, as shown by a system safety assessment, would lead to a failure of system function resulting in a minor failure condition for the aircraft.

e. Level E - Software whose anomalous behaviour, as shown by a system safety assessment, would lead to a failure of system function with no consequences for the aircraft.

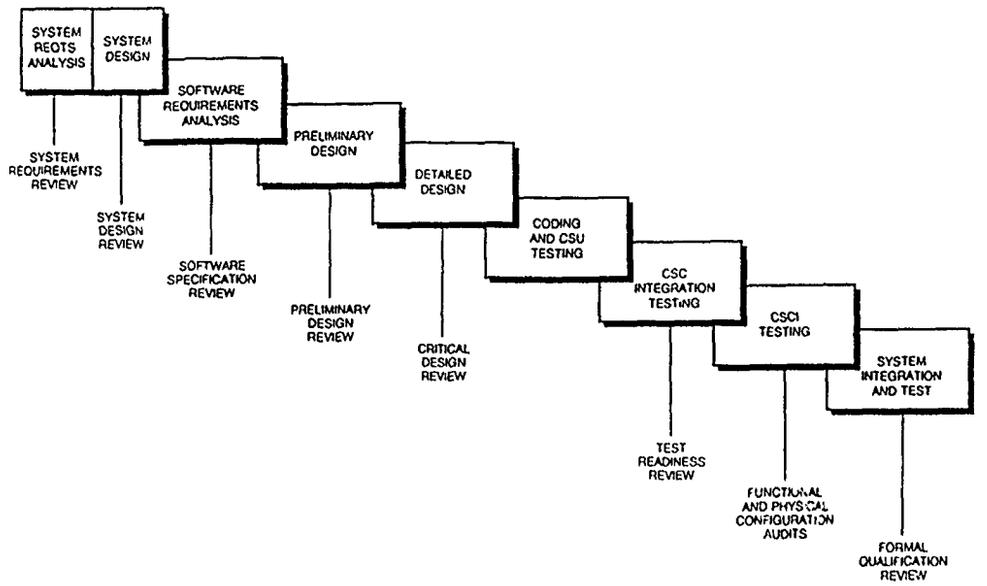


FIG.1 SOFTWARE DEVELOPMENT PROCESS DOD-STD-2167A

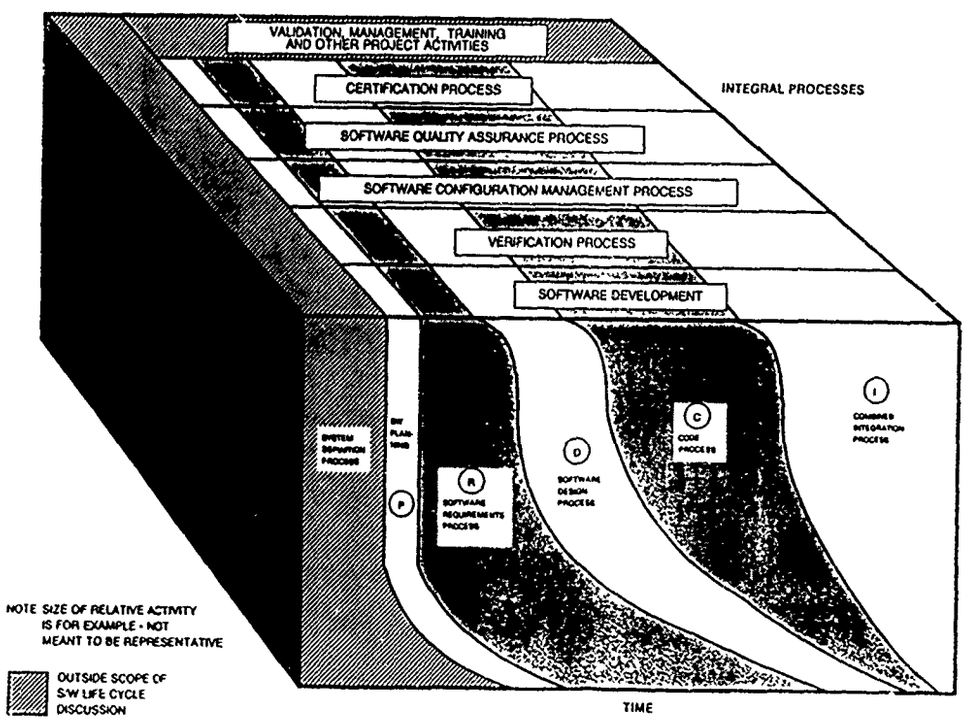


FIG.2 SOFTWARE LIFE CYCLE PROCESS MODEL RTCA DO-178B (DRAFT)

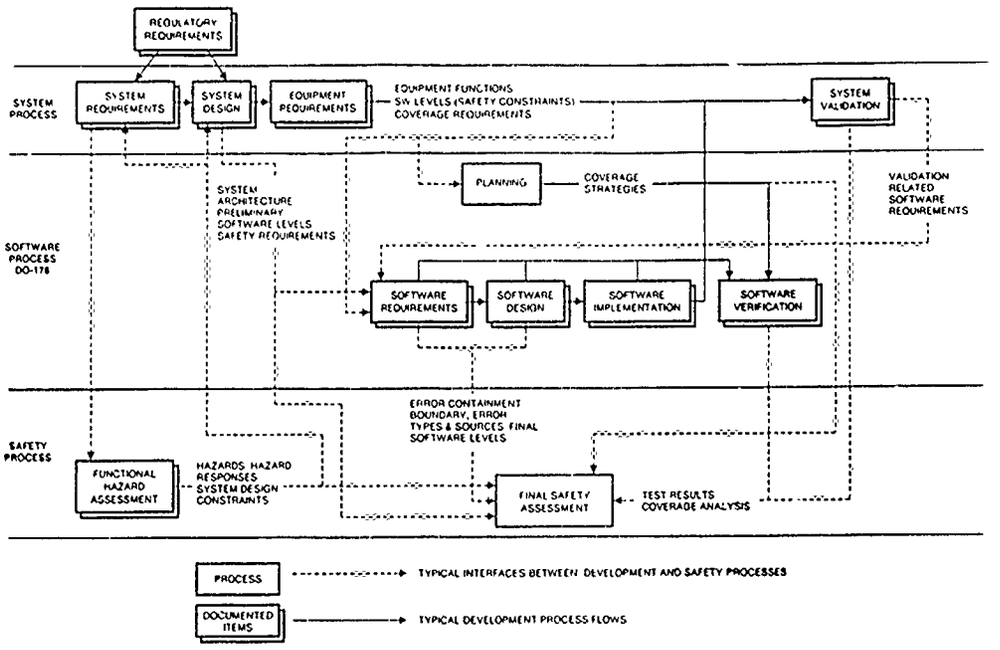
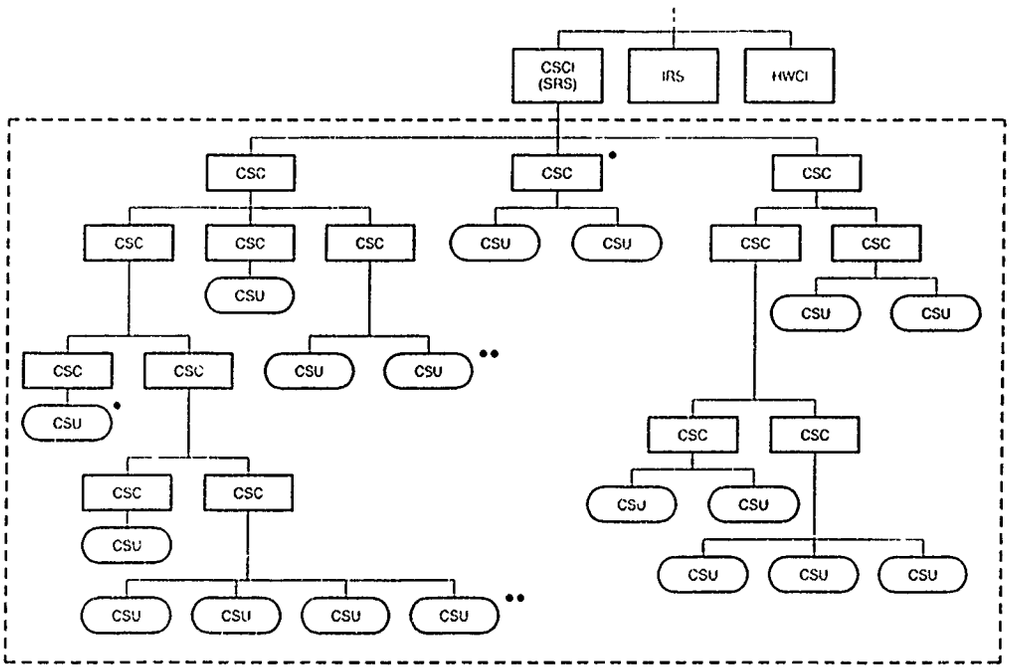


FIG.3 SYSTEM SAFETY PROCESS INTERACTION WITH SOFTWARE PROCESSES
RTCA DO-178B (DRAFT)



• NON DEVELOPMENTAL SOFTWARE
•• SAME CSU USED BY DIFFERENT CSCs

FIG.4 EXAMPLE OF CSCI DECOMPOSITION DOD-STD-2167A

DOCUMENT	REFERENCE NUMBER
PLANS	
Software Development Plan (SDP)	DI-MCCR-80030A
Software Test Plan (STP)	DI-MCCR-80014A
SOFTWARE DEVELOPMENT DOCUMENTATION	
System/Segment Specification (SSS)	DI-CMAN-80008A
System/Segment Design Document (SSDD)	DI-CMAN-80534
Interface Requirement Specification (IRS)	DI-MCCR-80026A
Software Requirement Specification (SRS)	DI-MCCR-80025A
Software Design Document (SDD)	DI-MCCR-80012A
Interface Design Document (IDD)	DI-MCCR-80027A
Software Test Description (STD)	DI-MCCR-80015A
Software Test Report (STR)	DI-MCCR-80017A
Software Product Specification (SPS)	DI-MCCR-80029A
CONFIGURATION CONTROL	
Version Description Document (VDD)	DI-MCCR-80013A
Engineering Change Proposal (ECP)	DI-CMAN-80639
Specification Change Notice (SCN)	DI-CMAN-80642
SUPPORT	
Computer System Operators Manual (CSOM)	DI-MCCR-80018A
Software Users Manual (SUM)	DI-MCCR-80019A
Software Programmers Manual (SPM)	DI-MCCR-80021A
Firmcare Support Manual (FSM)	DI-MCCR-80022A
Computer Resources Interface Support Document (CRISD)	DI-MCCR-80024A

FIG.5 DELIVERABLE DOCUMENTS DOD-STD-2167A

PLANS
- Software Aspects of Certification Plan
- Software Quality Assurance Plan
- Software Configuration Management Plan
- Software Verification Plan

SOFTWARE DEVELOPMENT
- System Requirements Document
- Software Requirements Document
- Software Design Description Document
- Source Code
- Software Verification Procedures and Results Document

CONFIGURATION MANAGEMENT
- Unit Configuration Identification Document
- System Configuration Identification Document

SUPPORT
- Support/Development System Configuration Document

STANDARDS
- Software Design Standards

CERTIFICATION
- Accomplishment Summary

FIG.6 SOFTWARE DEVELOPMENT DOCUMENTS RTCA DO-178 B (DRAFT)

REQUIREMENTS AND TRACEABILITY MANAGEMENT

Author: G M Cross, Marconi Underwater Systems Limited
Station Road, Weybridge, SURREY, KT15 2PW, ENGLAND

ABSTRACT

This paper explains the contribution of requirements traceability to the system development process in risk reduction and rework avoidance and the impact on all phases of project development from requirements capture through to customer acceptance and subsequent maintenance. By update of the traditional lifecycle model, the paper shows how the RTM[1] (Requirements and Traceability Management) product builds a system development environment addressing these issues and improving the benefits to Users of many of todays leading CASE tools by more effective integration, with a total lifecycle coverage.

Introduction and background to the work

Traditionally, the processes involved in System Development have relied heavily upon decisions made on the basis of experience and intuition. In particular, these decisions are predominantly made in the earliest, most critical phases of the project, where any errors have maximum cost impact. Furthermore, these critical decisions are often made arbitrarily and are rarely recorded in a formal manner. The traditional system development process therefore lacks traceability. The result of such an approach is a system which cannot easily be shown to meet the customer requirements.

As part of the investment by Marconi Underwater Systems Limited (MUSL) into producing high quality systems and software, MUSL have performed a careful analysis of the activities to be supported during the systems development lifecycle. Research started several years ago, when it became apparent to MUSL as an early adopter of CASE that some of these tools had serious deficiencies and were difficult to manage effectively over multiple projects. For example, poor integration meant excessive Engineer interaction to produce consolidated documentation, leading in turn to low maintainability as source data changed. In particular, the issues of technical control and management of traceability and configuration were poorly addressed, and lifecycle support was incomplete and fragmented.

In order to better understand these problems, MUSL produced a system development process model (SDPM) using the Yourdon method. This model considers system development as a series of transforming processes operating on the customer's requirements, to produce different representations of the system under development and resulting in the finished product offered for acceptance. An important result from the model is that management of the customer requirement, its detailed analysis and understanding, and traceability through to acceptance are key lifecycle activities.

Requirements Traceability

In the classic V-Diagram (Figure 1) we have historically underated the role of traceability in establishing early lifecycle verification as the design evolves.

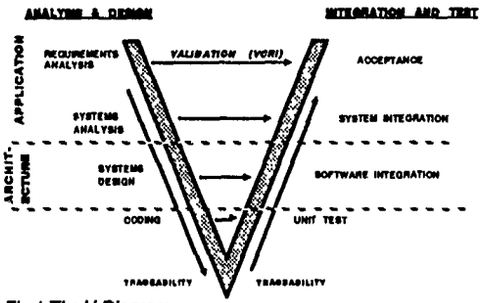


Fig 1 The V-Diagram

As we move out of the bottom of the V from code production we are all too often merely testing that the coded and integrated system accurately reproduces the errors lying undetected in the products of the preceding analysis and design phases. Then eventually, as we come to acceptance and we measure the system against the input requirement, the painful truth is finally revealed! It is a "standard result"[2] that almost two thirds of defects detected at integration and acceptance result from latent analysis errors.

Clearly what is needed is to establish a much earlier confidence in the quality of the requirement, followed by traceability into the analysis phase and beyond, and a comparison between the products of each phase to check for consistency. Traceability is "good common sense", and most technical managers will basically recognise that they use this approach informally in attempting to manage the risk in their projects, for example, as part of their design review activities It is also required by standard DOD-STD-2167A However, the full benefit can only be realised by rigorous application, and this demands effective tool support.

Traceability Toolset requirements

The results of the MUSL SDPM work also suggested that a standard model for traceability was impossible to agree, and that one ought to allow tailoring of the process model to optimise it for a given project. This in turn needs to be reflected by flexible configuration of the toolset.

In order to get the best return from traceability we need to examine the total system development process, as MUSL did with the SDPM, and then fabricate an IPSE where traceability is the underlying strategy for tool data integration. We thus identify the bridges which need to be built between the co-operating CASE tools in order to provide design traceability throughout the lifecycle. All of this has to be achieved of course, in the environment of a dynamically changing system requirement as the project proceeds. A successful implementation must provide facilities for:

- Total lifecycle support "Cradle to Grave"
- Initial requirement specification capture and subsequent configuration management
- Clarifying and refining poorly specified customer requirement statements

- Updating of Customer originated specifications, preserving Customer's format for meaningful dialogue
- Dynamic traceability, linking to all lifecycle phase products, and linking phase to phase
- Configurable traceability map to reflect local project needs
- Partitioning and managing designs, thus enabling sub-contractors to demonstrate compliancy at every phase
- Generation of compliancy reports supporting verification eg to DOD-STD-2167A
- Acceptance specification production facilities for system validation
- Audit trails of design history to support design review and maintenance
- Impact analysis for change management

These were some of the objectives of MUSL's RTM, and the rest of the paper explains the relevance and benefit of these objectives in an effective, high productivity total Systems Development Environment (SDE). The motivation behind the production of this environment and underlying method is to maximise support in the early phases of design for the scarce system designer resource, and to encourage rational and explicit decision-making by provision of a consistent set of guidelines for decision-making and recording of those decisions.

Components of an RTM based SDE

At the heart of RTM is the Project Database. This database holds information pertaining to all phases of the project, but focuses primarily upon the system requirements and the tracking of these through the project development cycle.

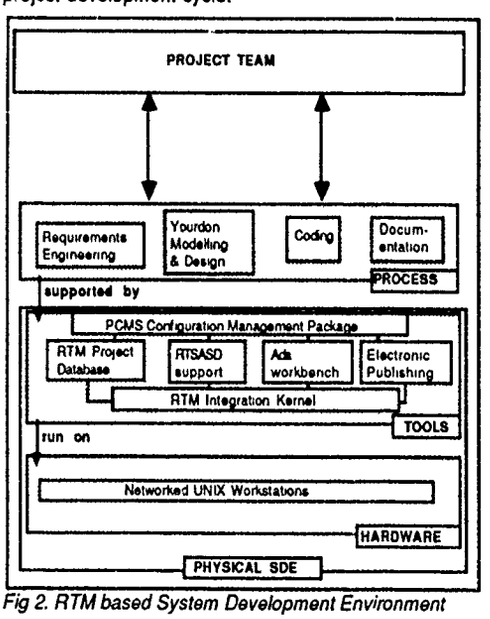


Fig 2. RTM based System Development Environment

This type of database is often referred to as a Verification Cross-Reference Index (VCRI), and is used to audit the compliancy of the project to the User requirement through the successive phases of development, culminating in Customer acceptance. Figure 2 shows the basic elements of a System Development Environment constructed around RTM, in the context of the software development tasks on a project. An upgradable architecture of UNIX workstations forms the platform for the tools.

Requirements Capture

In real world situations the requirements to be captured by RTM will come from various sources. Most commonly they will be presented as a Customer supplied document, or they may be presented as requirements assembled from a number of different documents. Under some circumstances they may be a set of derived requirements resulting from the reverse engineering of an existing system. The first step in the RTM process is to capture this information electronically. Those items which are not supplied by the Customer in an electronic form can be captured by scanning the Customer supplied document or if necessary typed directly into RTM afresh. Once this information is electronically captured the full facilities of RTM can be applied. To maintain and ensure the integrity of the original captured requirements they are made available to the RTM toolset in read-only mode.

Requirements Stripping

All documents which contain requirements relating to the proposed system are systematically transferred into the Project Database in a process known as requirement stripping. Typical examples of these documents would be documents defining the contractual standards, and documents defining system specific requirements. As each requirement statement is extracted from these documents and inserted into the database, the document identity and paragraph number from which it was extracted is recorded in the database. Any newly derived requirements which result from points of clarification or the like are documented and approved before being added to the project database as configuration items. After subsequent clarification and refinement as described below, it is then possible to reconstruct all of the original documents automatically to allow customer approval of the updates as valid interpretations of their needs. A browse facility in RTM enables users to scan the original requirements document either on a line by line basis or by using "string searches". The desired requirements text is interactively identified and transferred to a database, accompanied by a record of which section of the customer document it was extracted from.

Requirements Engineering

Once all the requirements have been extracted from the Customer's source documents it is necessary to examine and engineer them so that any ambiguities errors or duplicates are identified and addressed. Normally this activity would be performed by a small group of subject matter experts who are able to communicate directly with one another. The aim is to inject as much subject matter knowledge as possible into the requirements at the earliest stage, so that requirements are well defined before requirements subsets are passed on to the analysis teams. The analysis teams are then able to concentrate mainly on grouping and partitioning the requirements in a logical manner, using the specific subject matter knowledge which has already been injected. RTM provides facilities to enable the following requirements engineering functions to be performed whilst at all times maintaining the necessary links to provide an audit trail back to the functional requirements as stated by the customer:

One to One Substitution This is basically a simple edit of the requirement.

One to Many Substitution Allows a complex requirement to be broken down into its component parts thereby generating "child" requirements.

Many to One Substitution Allows duplicate or similar requirements to be focused down into a single requirements statement.

Clarification of Requirements Allows additional notes to be added (Engineers note pad) and associated with the requirement for use by analysts or designers later in the project's lifecycle. These notes are not normally produced or included in any automated reporting to the customer. The notes do not affect the text of the requirement.

Requirements Questions Provides for questions on specific requirements to be raised and fed back to the customer (points of clarification) with the updated specification.

It is not uncommon that a customer statement of functional requirements, for a medium sized system will generate many thousands of engineering requirements. It is easier and more efficient to manage large volumes of requirements if they are broken down into categorised subsets. Within RTM this is achieved by classifying the requirements by keywords:

Definition of Keywords Any character string can be defined as a keyword. Keywords can be linked to requirements either manually or by automatically searching through the requirements for the keyword character string. For example use of the string "Engineer" would result in all requirements where that string appears in the text being identified and linked to the keyword. Also a pseudonym text string can be used. In this case the requirements are searched for the pseudonym text string and those requirements linked to the keyword.

Keyword Reports Having grouped requirements together under keywords, reports can automatically be generated identifying which requirements are associated with what keyword.

Keyword Viewpoints Keywords can also be grouped together in sophisticated hierarchies allowing the selection and identification of requirements by a related group of keywords. This can be used as initial partitioning of the requirements as we move into the analysis phase.

Lifecycle Traceability

Having captured, understood and engineered the requirements, it is important to ensure that each requirement is correctly designed for and implemented, and that the impact of any future changes in the requirements is fully understood and traceable.

A high level view of the System Design Process model is shown at Fig 3 in a waterfall layout. It shows how RTM relates to the major project phases and how the major inputs to each phase are the outputs of the previous one. The outputs of each phase are baselines of the evolving design of the product, together with the design compliancy data against the requirements. From the figure it can be seen that RTM supports traceability through the entire project lifecycle from the initial capture of customer requirements through to delivery of the accepted systems and subsequent maintenance support.

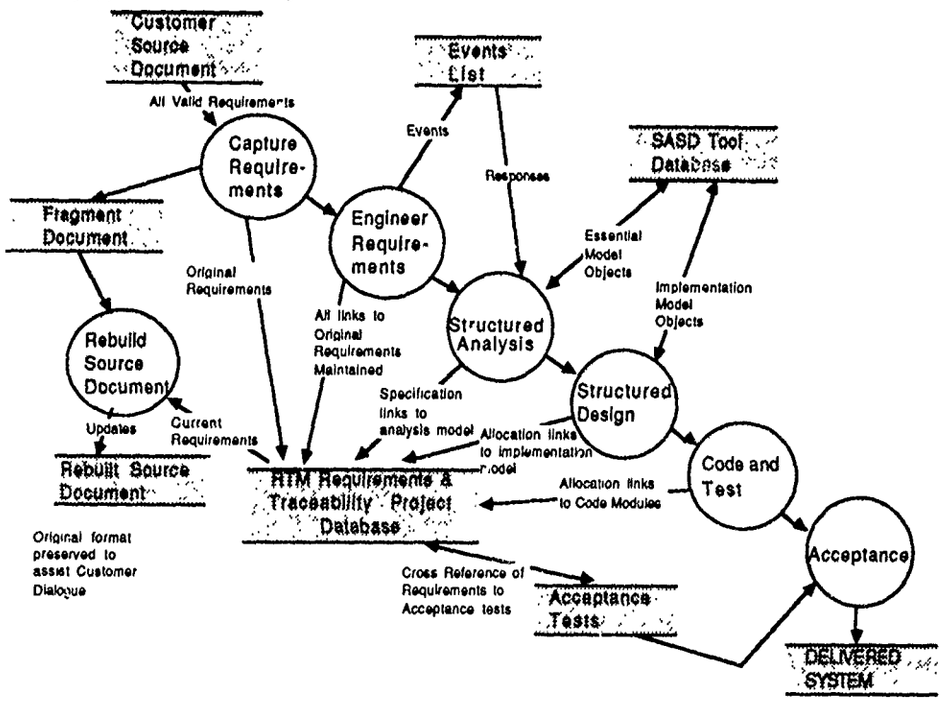


Fig 3. RTM project Lifecycle

With traceability links in place to all the products of design, it is then possible to see the impact of a specific requirements change down through analysis, design, implementation and testing.

The project phases are supported by the following tools:

Requirements	RTM
Analysis	Teamwork or StP
Design	Teamwork or StP
Coding	User Specified
CM	PCMS
Documentation	Framemaker or Interleaf

Analysis and Design

As we move into structured analysis and design (SASD) using for instance Yourdon, we first build an essential model which is equivalent to a Functional Baseline. This is a secure basis for progressing to the stage of allocating functions to hardware and software architectures in the implementation model. To provide traceability links to any elements of SASD models, integration modules are provided for industry standard SASD tools. Note that the analysis phase will typically cause some update to requirements, and with RTM, these can now be kept in track.

Because each object in the model is tied to the requirements it can be demonstrated that there is nothing superfluous or missing and full compliance can be readily demonstrated by a simple report from the Project Database.

In the design phase we are concerned with reviewing candidate solutions,to arrive at an optimum approach before producing the coded system. The candidate solutions are represented as "distortions" of the essential model. Distortion in this context means that extra functions may need to be added to establish communication between items of "off-the-shelf" equipment or software. Another example would be where an essential function is split between two processors for reasons of cost, performance, space or customer constraint. These constraints can be tied into the system design using RTM. Allocation of objects in the essential model can be mapped on to the implementation model objects to assure consistency of the two models.

Testing and Acceptance

A number of trials specifications are produced directly from the requirements in the database. These will detail the tests necessary to demonstrate to the satisfaction of the customer that a set of requirements have been fulfilled.

An acceptance test plan and record is drawn up which shows the manner in which the trials are conducted and records the outcome of those trials. Requirements are marked as having been accepted upon successful completion of their associated trial. A full audit of where the project is proving the implementation of requirements is available from the VCRI (Verification and Cross Referencing Index) report.

Traceability Schema Configuration

RTM provides facilities which allow entities and relationships to be defined to suit the particular requirements of a project and to be stored in the database. An example of such a schema to support a project is shown Figure 4.

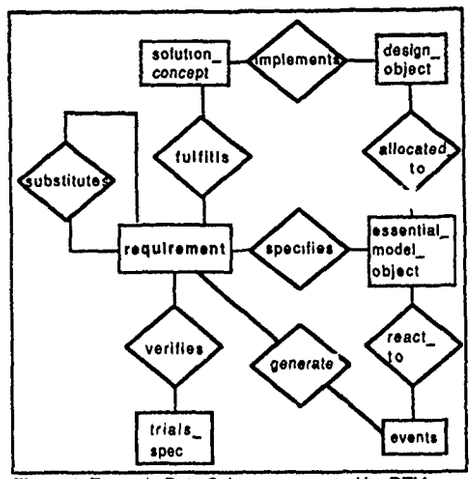


Figure 4- Example Data Schema supported by RTM

The user can define any object type which is required to support the project's lifecycle, for example, Event Lists, Essential Model Objects, Implementational Model Objects, Test specifications, etc. In addition the user can define any links (relationships) between the object types. These relationships can then be associated with a requirement. Where supported by the systems analysis and design tool in use, these objects and relationships can be described graphically by creating an entity relationship diagram from which information is captured and automatically entered into the controlling database.

Traceability to non integrated tools

Certain development tasks will be performed with tools which may not be integrated fully with RTM, for example reliability modelling. In such cases, the appropriate requirements are selected from the Oracle database using keyword classification. The teams responsible for addressing those requirements make compliancy statements against each of the selected requirements, indicating for each concept which requirements are fulfilled. In this way, full traceability is assured.

Impact Analysis

RTM provides standard reporting features which will identify objects which are linked to requirements, requirements which are linked to objects, and objects which are linked to other objects. These reports enable the impact of requests for changes to original requirements to be comprehensively analysed and assessed.

Compliancy Reporting and Audit

RTM also provides standard reports which will identify requirements not supported by analysis or implementational objects, which thus indicate deficiencies in the analysis or of the implementation design. Conversely reports can also be generated which will show analysis or implementation objects which do not support a specific requirement, which may indicate over design or the provision of functionality not requested by the customer. This also documents design decisions taken by engineers ensuring that project continuity is maintained even with staff turnover.

Automated Documentation

The ability to automatically generate meaningful, quality documentation in support of the requirements capture, design and implementation phases of the

lifecycle is very important. It improves project communication and enhances reporting and communication between the customer and project. When all the captured requirement have been engineered as described above, a complete report can be presented to the customer. This report will contain the customer's requirements mapped to the resultant engineered requirements, to any questions (points of clarification) as described above, and to the evolving design that satisfies the engineered requirements. These reports are fully automated, so that latest document versions can be produced with high productivity.

CONCLUSIONS

The Benefits of Requirements and Traceability Management

Throughout the project development lifecycle, the database will provide accurate and concise information relating to many aspects of the project:

- **Risk Management.** Resolving the ambiguities in the requirement specification as part of a meaningful dialogue with the customer, minimises the risk to both parties. Subsequently, through life traceability ensures we build the system we have contracted to build, and plan acceptance at the earliest stage possible.
- **Project Management.** It is possible to obtain an instant statement of the degree of compliancy in terms of those requirements which have been engineered, analysed, designed, implemented, tested and accepted. This provides the project manager with an objective indicator of the progress of the project.
- **Requirement Change Control.** When a requirement changes, it is possible immediately to determine which project tasks are affected either directly or indirectly, and how many hardware and software modules may have to be modified.
- **Testing.** When producing a test specification for a particular module, the Project Database will provide a list of all the requirements which the module should fulfil. The test can then be conducted on the basis of these requirements, and does not merely confirm that design errors have been faithfully reproduced by the implementation!
- **Integration/Acceptance** If during trials a requirement does not appear to have been implemented correctly, the Project Database will identify the module or modules which purport to implement the requirement in question, along with the analysis and design objects from which they were derived. This greatly reduces the extent and cost of the investigation which needs to be performed.
- **Documentation.** Because the majority of the system development tasks are performed using software tools, it is possible to automate the generation of high quality, consistent documentation to specified standards at the appropriate time.
- **Maintenance.** Several of the traditional difficulties encountered during maintenance are reduced due to the recording of traceability data in the project database. This reduces reliance on project experts, and allows areas likely to be affected by proposed changes to be more easily identified.

References

1. RTM (Requirements and Traceability Management) is a CASE tool developed by GEC-Marconi & MAIT Limited, and distributed in Europe by SQL Systems International of Harlow, Essex, England.

2. B. Boehm, IEEE Trans Software Eng. March 1975, P125.

COPROCESSOR SUPPORT FOR REAL-TIME ADA

by
R. K. Page
Senior Software Technologist
Naval Weapons Center
Code 3922
China Lake, CA 93555-6001
USA

SUMMARY

The purpose of this paper is to propose the basic elements of a real-time clock that would be suitable for use with the tasking mechanism of the Ada programming language and other real-time concurrency management systems. A real-time application needs such a clock for several reasons:

1. To relieve the processor of some of the overhead burden of time and task management.

2. To provide adequate granularity for the representation of time.

3. To provide sufficient range for the representation of time (References 1 and 2).

This paper also suggests a more complete solution to the overhead problem—move both the clock and the task scheduling functions normally implemented in software into a concurrency management coprocessor.

BACKGROUND

One of the main purposes envisioned for the Ada language was the programming of real-time embedded systems (Reference 2). A real-time system is a system containing real-time tasks. A real-time task is a task that has timing constraints (e.g., a deadline). Timing constraints may be hard or soft. A hard timing constraint must always be met (otherwise the system would fail). A soft timing constraint should be met if possible, but missing the constraint does not cause system failure. Fundamental to real-time systems is the concept of time. Real-time systems operate in an environment of severe timing constraints, with hard deadlines imposed on computations and input/output (I/O).

Some applications require very high periodic processor loads. In Equation 1, the parameter $U(n)$ represents processor utilization. (When the utilization is 1, the processor is 100% utilized.)

$$\sum_{i=1}^{n} \frac{C_i}{T_i} = U(n) \qquad (1)$$

where C_i and T_i are the execution time (including overhead) and period of task t_i, respectively. It is clear that processor utilization increases with either increasing compute time or decreasing period. If an overhead function (e.g., rendezvous) has a short compute time, a sufficiently short period may yield 100% utilization. For this reason, missile applications, with their short periods, require extremely short compute time for both application and overhead functions. High rates in the application may also require a finer resolution in the representation of time and, consequently, require a higher overhead cost to maintain the representation of time.

Because of inadequate hardware support and a marketing imperative to cover the largest set of target computers, existing Ada run-time systems are software intensive. As a result, a large portion of the time available for the application on the processor must be spent updating the real-time clock, managing the various scheduling queues, and scheduling tasks. Many hard real-time applications, such as missiles and robotics (Reference 3), generally use all of the time that the processor provides. If a significant portion of the processor's time would be devoted to managing Ada tasking, it could make the difference between using Ada tasking and writing a custom concurrency management system or a cyclic executive.

Some specific overhead functions that may require a large percentage of the processor's time are related to time and scheduling. For one Ada implementation, I found that the interrupt required to support package CALENDAR's notion of time-of-day and the delay statement, with 0.1 millisecond granularity imposed by the application, required greater than 30% of the processor's time. Addressing time management with software alone can consume a significant percentage of the total processor utilization. As task rates, tolerances on task rates (Reference 4), and the application utilization of the processor become more severe, a solution must be found to free more time for applications. One author (Reference 4) suggests, "With the ongoing interchanges in hardware/software trade-offs for improved performance, the future may reveal specialized hardware to alleviate some of these (real-time performance) problems." Although the importance of hardware support was recognized early by some (Reference 5), general recognition of this need is growing slowly in the real-time community as evidenced by papers appearing in journals (References 6 and 7), by papers presented at real-time workshops (Reference 1), and by some products directed toward real-time (Reference 8)

TIME MANAGEMENT COPROCESSOR

Fundamental to the management of tasks are data structures, such as delay queues and ready queues, and functions, such as reading time, setting alarms, and queue management. One way to reduce the time spent by the processor on overhead functions is to off-load the bulk of the run-time system's time management overhead onto a coprocessor. The time management coprocessor would be a hardware clock that would not need to interrupt the processor at a high rate to provide a fine granularity software clock. Because this device would contain a delay queue, the run-time would not need to implement or maintain a delay queue. The coprocessor would contain a priority-sorted ready queue and would not need to interrupt the processor to signal an expired delay unless the delayed task has a higher priority than the currently active task. For example, if a delay interval is requested, the interval would be added in the coprocessor to the current time, and the event identifier and resulting time would be placed automatically on the *Delay* queue. Upon expiration of the delay, the event identifier and priority would be moved automatically to the priority-sorted *Ready* queue. These capabilities would replace the software delay queue and some ready queue management with simple memory references. They could also reduce the number of interrupts required to support overhead functions.

MAJOR COMPONENTS

Figure 1 depicts the major functional blocks and general interfaces required for the time management coprocessor.

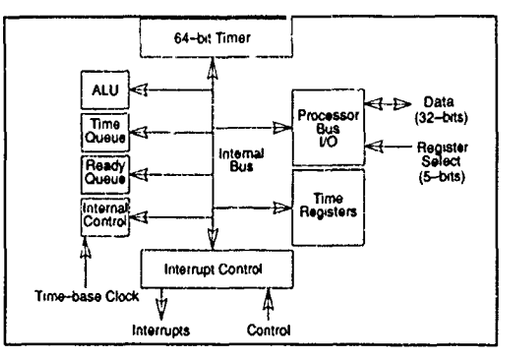


Figure 1. Time Management Coprocessor.

The major components of the device would be the following:

64-bit Timer—The least significant bit represents 1 nanosecond. The count represents the number of nanoseconds since 00:00, 1 January of some user-defined year (see the section titled Representation of Time) or a count of nanoseconds since initialization of the timer.

Time Queue—The *Time* queue is a time-ordered queue that contains the scheduled event time, an event identifier (usually the task identifier), and the task priority (see the section titled Periodic Execution).

Ready Queue—The *Ready* queue is a priority-ordered queue of events for which the scheduled time has expired. This queue contains the event identifier and the task priority for each expired event.

Time Registers—This section contains all the registers required for operations within the device:

CURRENT PRIORITY—set by the run-time system to the priority of the currently executing task (see the section titled Delay Scheduling). Because this register is, as are all registers, memory mapped, the time required to update the priority is the same as writing an integer to a memory location during a task switch.

YEAR_NUMBER—as derived from the 64-bit representation of time (see the section titled Representation of Time).

MONTH_NUMBER—as derived from the 64-bit representation of time (see the section titled Representation of Time).

DAY_NUMBER—as derived from the 64-bit representation of time (see the section titled Representation of Time).

SECONDS_NUMBER—as derived from the 64-bit representation of time (see the section titled Representation of Time).

NANOSECONDS_NUMBER—as derived from the 64-bit representation of time (see the section titled Representation of Time).

TIME_NUMBER—for the 64-bit representation of time (see the section titled Representation of Time).

STATUS REGISTER—indicates overflow, underflow, sign, zero, a negative or zero delay, or a delay_until scheduled for a time that has passed (see the sections titled Time-Related Operations and Delay Scheduling).

COMMAND REGISTER—commands to control and set the mode of the coprocessor are written to this register by the processor (see the section titled Time Management Coprocessor Functions).

SAMPLE CLOCK REGISTER—for the current count when commanded under hardware control (see the section titled Initial Setting).

UPDATE REGISTER—for a predetermined number of the least significant bits to replace the corresponding bits in the current time. This is done under hardware control (see the section titled Clock Synchronization).

ALU—An arithmetic/logic unit (ALU) is used for performing arithmetic time operations (see the section titled Time-Related Operations).

Interrupt Control—Interrupts the processor if the event at the top of the *Ready* queue is of a higher priority than the priority in the CURRENT PRIORITY register.

Internal Control—Executes commands and controls the components of the time management coprocessor.

Host Processor Bus I/O—Controls communications with the host processor.

REPRESENTATION OF TIME

With the time management coprocessor, time would be internally represented as a 64-bit count of nanoseconds. There are two related reasons for the coprocessor to use a count of nanoseconds. First, to provide for a monotonic clock during distributed system clock synchronization, a finer timer granularity may be required than for any application task (see the section titled Clock Synchronization). Second, although nanosecond granularity is not required for most applications today, I believe that any attempt at addressing timing issues should address longer term possibilities.

The time may come, with faster applications (aircraft, missiles, etc.) and higher speed processors (gallium arsenide (GaAs)-based processors (References 9 and 10) for example), that nanosecond granularity may be required. However, the device could be designed to increment only those sub-second bits that are appropriate for current technology. For instance, if the hardware technology supported a 1-megahertz clock rate (1 microsecond granularity), the ten least significant bits of the nanosecond field from an applications point of view would always be zero on output and "don't care" on input (see the section titled Clock Synchronization). This 64-bit representation would support over 290 years of nanoseconds, which should be enough for most Ada applications. The run-time system or application could read or write directly to the device using the 64-bit (nanosecond count) representation of time. However, the most common representation would be the following:

nanoseconds—representing nanoseconds less than a second (30 bits minimum)
seconds—representing second of the day (17 bits minimum)
days—representing day of the month (5 bits minimum)
months—representing month of the year (4 bits minimum)
years—representing absolute or relative years (10 bits)

Each of these would be an integer value written to a separate address on this memory-mapped device. When the derived representation is written to the appropriate address, the device would internally convert the derived representation to the 64-bit nanosecond count representation for use in setting the clock, setting a delay, or setting another of the time-related functions. Time representation could also be read from the device in the derived (by automatic conversion) or the 64-bit nanosecond count (direct) format.

TIME MANAGEMENT COPROCESSOR FUNCTIONS

Several functions are required by this device to support the run-time system. The following is not intended to be an exhaustive list, but only to represent some basic commands.

SETTING THE CLOCK

Initial Setting

To set the clock, the user would write the current time in either the internal or the derived format. It would be necessary to enter only the parameters of time that are of interest to the application. For example, in a missile application, time of flight and not absolute time is the more appropriate representation. The user could, therefore, initialize the clock to zero time at the start of the mission. For shipboard-, ground-, or space-based applications, the user may require the full representation of time: a nanosecond count from a user-specified time. Whether the time used is relative or absolute would be determined by a mode command issued to the coprocessor.

Clock Synchronization

In distributed applications, the need for clock synchronization becomes apparent. Clock synchronization takes two forms: start-up synchronization and correction for drift. If the application required setting the clock to the value of a master clock, this could be done in one or two stages, depending upon the application and its required accuracy. The first stage would involve copying the time from a "standard" clock to the coprocessor and then issuing the set clock command. (This first stage should be nonpreemptable.) The second stage is the synchronization process.

Start-up synchronization and correction for drift (Reference 4) could be accomplished in a similar manner. If time were sensed to have drifted, the user would have the ability to correct the time with three levels of processor interaction, depending upon the accuracy required.

Full processor involvement—the processor is a critical element in the accuracy of the synchronization process. An example of this type of synchronization is given in the previous paragraph.

Partial processor involvement—the processor is involved; however, processor timing is not critical to the accuracy of the synchronization process.

Processor independent—the processor is not involved in the synchronization process on a continuing real-time basis.

One method for partial processor involvement would involve a hardware-based signal for the coprocessor. The coprocessor would be equipped with a *sample clock* input that would force the clock to immediately copy the counter to the SAMPLE CLOCK register. The processor would next copy the SAMPLE CLOCK register of a master clock to the coprocessor. The processor would then issue the subtract and update command. This command would cause the coprocessor to subtract the master clock time sample from the value in the SAMPLE CLOCK register and then add the result to the current time. This method could incur an unacceptably large processor overhead. Also, if the clock were being used for timing measurements, the results would not be acceptable.

A method to implement the third form of synchronization would involve the processor only during system initialization. During initialization, the processor would enter a predetermined number of the least significant bits into the UPDATE register. At an appropriate interval, the master clock would issue a hardware-based update command to the coprocessor. The coprocessor, upon receipt of this command, would copy the UPDATE register into the least significant bits of the current time. If the bits set in this process are less significant than the granularity required by the application, time monotonicity is assured.

These are some of the simpler clock synchronization schemes that could be used. However, the device should be compatible with approaches taken in bus standards, such as Futurebus+ (Reference 11).

Delay Scheduling

To implement the current Ada delay statement, the run-time system would write the delay time (in either representation of the section titled Representation of Time), the task priority, and the task identifier (TID) and then write the command to set delay. Internally, the device would add the delay to the current time and then enter the absolute expiration time into the *Time* queue along with the priority and the TID associated with the event. If the requested delay is zero or negative, this will be flagged in the STATUS REGISTER (see the section titled Major Components), and the information for the task will be placed directly in the *Ready* queue.

For each count of the 64-bit timer (which could represent 1 nanosecond for a 1-gigahertz time-base clock, 1 microsecond for a 1-megahertz time-base oscillator, etc.), the device would compare the earliest time on the *Time* queue with the current time. When the current time is greater than or equal to the scheduled time, the priority and TID would be transferred to the *Ready* queue.

The priority and TID of the highest priority runnable task would always be readable by the run-time system. If the run-time system has written a value to the CURRENT PRIORITY register, the device will wait until the highest priority runnable task has a priority greater than the CURRENT PRIORITY register before it generates an interrupt. To disable the interrupt, the run-time system would write the value of PRIORITY'LAST to the CURRENT PRIORITY register. To always be interrupted when a delay expires, the run-time system would write the value of PRIORITY'FIRST to the CURRENT PRIORITY register.

A delay_until statement (Reference 12) operates in a similar fashion. The difference is that the absolute time for expiration is entered when a set delay_until command is issued to the device. If the scheduled time has passed, this will be flagged in the STATUS REGISTER (see the section titled Major Components), and the information for the task will be placed directly in the *Ready* queue.

Other Scheduling Operations

If a task required removal from either the *Ready* queue or the *Time* queue when a task terminates, is aborted, or completes a timed entry call, the run-time system would write the TID to the device and then write a coprocessor command to delete the *Time* queue entry or to delete the *Ready* queue entry.

If a task priority must be changed, the run-time system would write the TID, the new priority, and the change priority command to the coprocessor. This would change the priority of all occurrences of that task in both queues.

Time-Related Operations

The following functions would be available from the device:

- Convert a 64-bit representation, written by the run-time system or application program, to YEAR_NUMBER, MONTH_NUMBER, DAY_NUMBER, SECONDS_NUMBER, and NANOSECONDS_NUMBER (Reference 2).

- Convert YEAR_NUMBER, MONTH_NUMBER, DAY_NUMBER, SECONDS_NUMBER, and NANOSECONDS_NUMBER to the 64-bit representation.

- Add an interval to the current time, in either the derived format or the 64-bit format, to

the current time. The output would be available in either the derived format or the 64-bit format.

- Subtract the clock from a time in the future. The input would be in either the derived format or the 64-bit format. The output would be available in either the derived format or the 64-bit format. This operation would also allow comparison of the current time with a given time with the result available from the STATUS REGISTER (see the section titled Major Components).

OTHER OPERATIONS

Watchdog or Alarm Functions

A watchdog or alarm function would be treated like a delay statement or delay_until statement by the coprocessor; however, if no TID would normally be associated with the watchdog or alarm function, the run-time system would fabricate a special identifier to flag the function. The advantage of having the watchdog function as part of the same device is that the priority of the watchdog expiration would be sorted with the priorities of the tasks in the *Ready* queue. Because watchdog priorities are combined with task priorities, the application will not have a high priority event interrupted by a low priority event. A user, for example, may not want the "fight fire" task interrupted by the "popcorn ready" interrupt.

Periodic Execution

Through the addition of another field and more control logic in the *Time* queue, automatic rescheduling of periodic tasks could be supported. The additional field in the *Time* queue would represent the period of the associated task. On expiration, the TID and priority would be moved to the *Ready* queue. The device would, on finding a nonzero period field, add the period to the expired delay time for the task and then enter the new time, the priority, and the TID into the *Time* queue. If the periodic task terminated, the run-time system would write the appropriate delete command to the coprocessor (see the section titled Other Operations).

IMPLEMENTATION OPTIONS

Reduced Capability Implementations

The discussion here has centered on producing a device that has full capability to support run-time system time-related functions. While this is desirable, a device with less functionality could also improve system performance. The basic timer elements required to support the run-time system (Figure 2) are the 64-bit timer, an expiration time register, and a time comparator. With this basic device, the run-time system would keep a time-sorted queue of time events and write each event

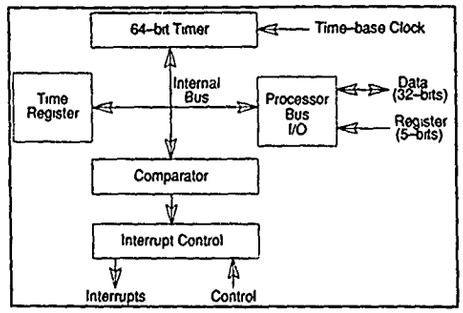


Figure 2. Time Management Coprocessor Essentials.

time to the device immediately after the current timer event expires or as a new event precedes the current timer event. This simple implementation would require an interrupt only when a scheduled event expired. The run-time system or the application could read the current time.

While the coprocessor's requirements could be implemented in today's gate array technology (Reference 13), it is unlikely that a 1-nanosecond clock granularity will be required for the processors being designed into systems today. A granularity of 0.1 microsecond should be easily achieved and should cover the majority of applications today. A processor based upon VHSIC Phase II or GaAs technology could support 0.01 microsecond or less granularity because of their 100+ megahertz clock rates (References 9 and 13).

Other pieces of the full coprocessor may be added to the basic timer depending upon their relative contribution to run-time system overhead. If the operations involved in combining various time units into the nanosecond count and deriving those time units from the nanosecond count are a larger burden than queue management, they would then be implemented. Coprocessor implementation with an ALU could supersede implementation of a comparator for signaling the expiration of a scheduled event. It is also clear that many more run-time system functions could be added to this type of coprocessor and result in reduced overhead in the run-time system (Reference 6).

Coprocessor Versus On-Chip

One question must be addressed to ensure compiler vendor support and applicability to the broadest number of systems. Should the time management device be implemented as a coprocessor in a separate package from the processor (off-chip) or integrated with the processor on a single chip? I believe the device should initially be implemented off-chip.

The following are some technical advantages of on-chip implementation.

- Signal propagation delays involved in communicating with an off-chip device may be ten times those of an on-chip device.

- Less power would be required by an on-chip device.
- A smaller footprint would be required for an on-chip device.
- Potential performance gains exist from integrating the timer registers and commands with those of the processor.

The technical advantages of on-chip devices must wait until experience with them is gained and the market for them is developed. The following are advantages of the off-chip time management coprocessor.

- The coprocessor device can be designed to work with a variety of processors currently in use for real-time applications. This flexibility would provide a broader market base for the device.
- The selection of an appropriate timer could be independent of the selection of an appropriate processor.

Implementing the off-chip coprocessor first and then following that with development of the on-chip device was the approach taken successfully with the floating-point coprocessor and memory management units (see the section titled Summary and Conclusions).

SUMMARY AND CONCLUSIONS

In the past, we have seen software floating-point arithmetic performed to the detriment of real-time system performance. Now we have hardware floating-point coprocessors. In the past, we have had software-intensive memory management systems. We now have hardware memory management coprocessors. Both of these devices have relieved the processor of a computational burden shared by many applications. At the present time, we have software-intensive "time management" as evidenced in the Ada run-time systems where software manages the delay queues, ready queues, time scheduled events, etc. We need similar hardware support for real-time performance to make Ada viable for a broader range of real-time systems.

A time management coprocessor could be easily implemented with today's gate array technology, although it would probably be limited to 0.1 microsecond granularity. The main issue in the design of the coprocessor is to ensure that it interfaces properly with a broad range of processors (Reference 13).

A time management coprocessor should be only a first step in moving more overhead functions into the hardware. Implementations of this device could range from the basic device in Figure 2 to the more complete device shown in Figure 1. While this coprocessor has the potential of relieving the processor of some overhead, a more complete solution is required. The more complete solution includes moving time management functions, scheduling algorithms, and other concurrency management functions from the run-time system and compiler-generated code into a concurrency management coprocessor (References 4, 6, and 10). The concurrency management coprocessor has been demonstrated by Lund University as a practical and feasible way to build computer systems. This solution will never be applied if the design of computing systems is maintained as separate hardware and software entities.

We are now entering an era that requires and can support a new view of system design—a view inspired by the need for higher speed and more correct complex systems. The foundation for this new view is based upon powerful hardware and software tools that have developed along similar lines (e.g., silicon compilers, VHDL, software compilers, Ada). What appears to be lacking are methods to support the work and system engineers with appropriate expertise (Reference 10). The work discussed here is dramatic evidence of the potential of taking a new view of system design.

REFERENCES

1. N. H. Weiderman. "Real-Time Programmers Don't Use Calendars," Third International Workshop on Real Time Ada Issues, 1989.

2. U.S. Department of Defense. *Reference Manual for the Ada Programming Language.* January 1983. (ANSI/MIL-STD 1815A.)

3. Thomas E. Bihari. "Current Issues in the Development of Real-Time Control Software," IEEE Computer Society Real-Time Systems Newsletter, 15, 2, 1989, pp. 1-5.

4. Robert K. Page. Naval Weapons Center. *Coprocessor Support for Real-time Ada.* March 1991. (NWC TM 6958.)

5. M. Ganapathi and G. O. Mendal. "Issues in Ada Compiler Technology," *Computer*, 22, 2, February, 1989, pp. 52-60.

6. Paul N. Hilfinger. "Implementation Strategies for Ada Tasking Idioms," *Proceedings of the Ada TEC Conference on Ada,* 6-8 October 1982.

7. Joachim Roos. "A Real-Time Support Processor for Ada Tasking," *ASPLOS-III Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE Computer Society Press (Order Number 1936), 1989.

8. R. A. Volz and T. N. Mudge. "Instruction Level Timing Mechanisms for Accurate Real-Time Task Scheduling," *IEEE Transactions on Computers*, C-36, 8, August 1987, pp. 988-993.

9. Intel Corporation. 80960MC Programmer's Reference Manual. IC, P.O. Box 58130, Santa Clara, CA 95052-8130, 1988.

10. R. Weiss. "GaAs Bears Fruit at TI," *Electronic Engineering Times*, 514, 28 November, 1988, pp. 53-54.

11. W. Helbig and V. Milutinovic. "A DCFLE-D-MESFET GaAs Experimental RISC Machine," *IEEE Transactions on Computers*, C-38, 2, February, 1989, pp. 263- 275.

12. R. A. Volz, D. Wilcox, and L. Sha. "Position Paper on the Global Clock for the Futurebus+." Draft, 1989, p 4.

13. L. Sha and J. B. Goodenough. Software Engineering Institute. *Real-Time Scheduling Theory and Ada*, April 1989. (Technical Report CMU/SEI-89-TR-14.)

14. V. Anderson, Naval Weapons Center. Code 3649 E-Mail communication, 1989.

15. L. Philipson. "Multilevel Design and Verification of Hardware/Software Systems," *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 3, June 1990, pp. 714-719.

16. R. Woolnough. "Chip Accelerates Ada," *Electronic Engineering Times*, 9 October 1989, pp. 20 and 26.

17. Vectron Laboratories, Inc. Crystal Oscillators 1987. VLI, 166 Glover Ave., Norwalk, CT, 06850, 1987.

ATELIER DE DÉVELOPPEMENT DE LOGICIELS DE PILOTAGE - GUIDAGE

par
D.CAIGNAULT, S.GABISON, JL.LEBRUN
SEXTANT Avionique
Aérodrome de Villacoublay
78141 Vélizy Cedex
FRANCE

0. Résumé

Les équipements de pilotage et guidage développés par SEXTANT Avionique possèdent des architectures de plus en plus complexes et la part du logiciel est sans cesse croissante.
Pour répondre à ces nécessités, SEXTANT Avionique a mis en oeuvre un atelier de développement de logiciels de pilotage et guidage. Cet atelier est constitué d'outils et de passerelles communicantes assurant la cohérence sur tout le cycle du développement.
VISA (Validation Interactive de Spécifications Avioniques) permet la formulation des spécifications et offre la possibilité de les rendre exécutables, pour d'une part, évaluer très tôt leur comportement dynamique (maquettage) et d'autre part, vérifier le comportement en temps réel de l'équipement spécifié (prototypage).
La conception et la réalisation du logiciel sont facilitées par l'emploi d'outils de génération automatique de code de l'atelier :

- **GALA** (Génération Automatique de Logiciel Avionique) génère le code exécutable des fonctions de pilotage et guidage à partir des spécifications détaillées décrites dans le langage graphique sous VISA.
- **GALI** (Génération Automatique de Logiciel d'Interface) génère le code exécutable de traitement des entrées/sorties décrites dans le dictionnaire de données sous VISA.

L'intégration et la validation s'effectuent en plusieurs phases. Le Banc de Validation Avionique (**BVA**) a pour objectif de simplifier la mise en oeuvre et l'exploitation des résultats de ces tests.
Les outils de l'atelier sont placés dans une structure d'accueil **PALAS** (Production Assistée de Logiciel d'Application Structurée) assurant la cohérence de l'ensemble.

1. Introduction

La conduite du vol est un des métiers de base de SEXTANT Avionique.

Dans le domaine de la conduite du vol civil, la participation de SEXTANT Avionique à la gamme Airbus, commencée au tout début des années 70, a été marquée par certaines "premières" qui ont constitué des avancées technologiques marquantes:

- systèmes d'atterrissage automatique tout temps avec une hauteur de décision (HD) progressivement amenée à zéro,
- développement sur A300-B4 du premier FFCC (Facing Forward Crew Cockpit), pilotage à deux, basé sur un système de conduite du vol numérique,
- commandes de vol électriques d'abord sur l'A310 pour les gouvernes secondaires, puis globalement sur l'A320,
- intégration des fonctions de conduite et de gestion du vol pour A320, puis A340 dans un seul calculateur.

Dans le domaine de la conduite du vol militaire, SEXTANT Avionique participe à la définition et à la réalisation des pilotes automatiques de tous les avions d'armes français Les derniers équipements fournis concernent les différentes versions du Mirage 2000 (Défense aérienne, Export, N et D), les avions ATL-2 (patrouilleur maritime), le C135FR (ravitailleur en vol de l'armée de l'air française) et l'hélicoptère franco-allemand TIGRE.
Pour le RAFALE, SEXTANT Avionique conduit des travaux sur l'approche et l'appontage automatique, sur l'approche et l'atterrissage sur terrain de fortune, fonctions de pilotage automatique intégrées dans le CET (Calculateur d'Elaboration de Trajectoires).

Ces équipements se caractérisent :

- par des architectures fonctionnelles et matérielles de plus en plus complexes,
- par une numérisation quasi-totale, induisant des volumes de logiciel en accroissement permanent,
- par des développements à délai constant, impliquant une maîtrise du cycle de développement au travers d'outils performants et d'une organisation industrielle adéquate.

Tous ces programmes, développés de plus en plus en large partenariat, reposent sur des technologies de pointe, des logiciels et des équipements à haut niveau de sécurité.

Cette communication insiste sur les exigences en matière de qualité de logiciels qui ne peuvent être assurées qu'au travers d'une méthodologie rigoureuse couvrant le cycle de vie complet. La formalisation de cette méthodologie à l'aide d'outils informatiques est un atout supplémentaire à la satisfaction de ces exigences.

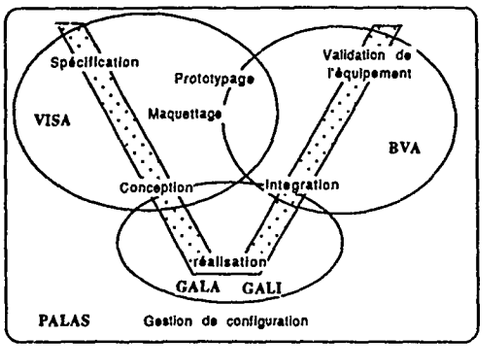
La communication décrit dans l'ordre habituel du cycle de développement du produit l'ensemble des méthodes et outils mis en place par la société SEXTANT Avionique pour la spécification, la conception, la réalisation, l'intégration et la validation des équipements de conduite du vol et souligne la cohérence de l'ensemble au travers de passerelles communicantes.

Les activités concernées par l'élaboration d'équipements avioniques sont réparties classiquement dans un cycle en V et représentent successivement :

- la formulation des exigences (spécification) et la recherche des solutions permettant de les satisfaire (conception),
- la réalisation (matériel et logiciel) implémentant les solutions,
- l'intégration et la validation fonctionnelle globale de l'équipement.

La démarche entreprise a pour but d'arriver au niveau de détail final selon une méthode d'analyse descendante, en assurant une traçabilité dans le développement et de lier les différentes phases au travers d'outils dédiés à chaque tâche et compatibles entre eux :

- **VISA** (Validation Interactive de Spécifications Avioniques) pour les tâches de spécification ,
- **GALA** (Génération Automatique de Logiciel Avionique) et **GALI** (Génération Automatique de Logiciel d'Interface) pour les tâches de conception et de réalisation du logiciel,
- le Banc de Validation Avionique (**BVA**) pour les tâches d'intégration et de validation fonctionnelle,
- PALAS ™ (Production Assistée de Logiciel d'Application Structuré) pour l'organisation du projet et la gestion de configuration.



ATELIER DE DEVELOPPEMENT DE LOGICIELS DE PILOTAGE GUIDAGE

2 Spécification, maquettage, prototypage

2.1 Spécification

VISA est un outil développé par SEXTANT Avionique conçu dans le cadre d'un métier particulier dont le savoir-faire est parfaitement stabilisé : le developpement de logiciels de pilotage-guidage.Visa est utilisé pour l'élaboration de la spécification globale puis detaillée.Il a été élaboré avec le souci de satisfaire aux contraintes imposées par les collaborations industrielles multi-partenaires :

- diversité des méthodes et outils de rédaction du cahier des charges qui peut se présenter sous forme de spécification globale ou détaillée,
- partage des tâches et communication des spécifications organisées dans un cadre contractuel.

Les difficultés sont surmontées grâce à la modularité de l'outil et à un support méthodologique rigoureux.
VISA répond, de plus, aux besoins suivants :

- accroître la qualité et la fiabilité de la spécification produite au travers d'une description structurée et non ambigué,
- valider la spécification avant la phase de réalisation,
- intégrer la documentation .

Analyse descendante

La spécification, formulation des exigences, se décompose en plusieurs phases, avec un niveau de détail croissant. Ces phases aboutissent à des formulations différentes,mais cohérentes entre elles,du cahier des charges. .

La spécification globale s'appuie sur une méthode de décomposition fonctionnelle, héritée de la méthode SART, suivant le formalisme proposé par HATLEY. Le concepteur affine progressivement les fonctions nécessaires à la réalisation du cahier des charges et précise les flux de données associées. L'architecture générale du produit est ainsi mis en place et est utilisée pour construire une gestion de configuration (entités et liens de dépendance inter-entités) et initialiser les phases suivantes du développement.

Spécification détaillée

La phase de spécification détaillée est précédée d'une étude préliminaire de la structure des lois de pilotage et de la mise au point des solutions répondant au besoin. Cette tâche est réalisée avec l'aide d'outils d'AAO (Automatique Assistée par Ordinateur).

Les solutions sont ensuite formalisées grâce à un langage graphique très proche de celui de l'automaticien. Une bibliothèque de fonctions standardisées est disponible et comprend, entre autres, des fonctions de type arithmétique, trigonométrique, logique et des fonctions plus complexes telles que filtre, intégrateur, retard, moyenne, limiteur. Ce langage sera décrit plus précisément au §3.
Les solutions sont développées conformément au découpage fonctionnel et à la définition des flots de données, issus de la phase de spécification globale. Elles sont très finement détaillées afin de permettre un codage direct.
La représentation graphique de l'ensemble de la spécification est le vecteur privilégié de la communication grace à son formalisme clair, non ambigu et accessible à différents niveaux d'abstraction (décomposition hiérarchique descendante). Elle est rendue indispensable du fait de la diversité des intervenants.

Base de données

L'analyse du cahier des charges débouche, comme on l'a vu, sur l'élaboration d'une spécification globale, puis détaillée, finalisant ainsi la bonne compréhension du problème et l'assurance de la faisabilité théorique. Les informations susceptibles d'être utilisées dans plusieurs phases du développement (types, natures, destinations des flots de données, attributs des éléments caractéristiques) sont extraites de ces spécifications et sont ordonnées dans une base de données (BD), appelée BD interne, qui est exploitée pour assurer la cohérence de la spécification et apporter un complément pratique d'informations à celle-ci.

La spécification est parallèlement complétée par la définition des entrées/sorties de l'équipement, également ordonnées dans une deuxième BD, appelée BD externe. Son exploitation permet d'élaborer un document officiel, livré avec l'équipement pour en spécifier les connexions.

Ces deux BD sont utilisées conjointement et des relations permettent d'en croiser les informations.

Gestion et documentation intégrées

Il est indispensable de conserver à l'ensemble une cohérence constante, tant au niveau du contenu que de la configuration. En phase de développement, les enrichissements sont multiples et la charge de travail imposée par les tâches de gestion requiert une mobilisation importante. Aussi, il est apparu indispensable d'y associer un guide méthodologique précis, reposant sur les principes suivants :

- centralisation et transmission verticale des informations à travers l'exploitation de la BD interne, lors des différentes phases de spécification (contrôles croisés sur le contenu, analyse de cohérence et de complétude, répercussion des modifications, consultation ergonomique),
- gestion de configuration globale, définissant les liens de dépendance entre éléments de différentes phases.

De même, l'élaboration de la documentation, selon une norme précise, réclame la plus grande rigueur méthodologique et est avantageusement supportée par un outil de PAO (Publication Assistée par Ordinateur).

Ces diverses tâches font partie intégrante de l'atelier. Leur traitement en apparait alors moins contraignant, laissant aux intervenants plus de disponibilité pour les tâches afférentes à la recherche de solutions répondant aux exigences fonctionnelles.

Outils

Les méthodes présentées ci-dessus sont mises en oeuvre par des progiciels qui sont, chacun, dédiés à une partie distincte de la spécification. Chaque outil possède un contrôle intégré de cohérence et de syntaxe, de la partie qu'il est chargé de décrire, ainsi qu'une BD propre.
L'outil VISA possède une structure modulaire, dont les composants de base, pour la spécification, sont STP™ (sté IGL supportant SART), un éditeur graphique SAFIRS™ (sté ASSIGRAPH). Ces outils peuvent être remplacés par des outils équivalents pour satisfaire les besoins spéci-

fiques d'activités dans le cadre de coopérations industrielles.
L'outil de gestion de configuration (PALAS™ cf §5) et le gestionnaire des BD interne et externe (ORACLE™) assurent la cohérence de l'ensemble.
Ces outils sont intégrés dans une structure d'accueil à interface conviviale et l'ensemble constitue un atelier cohérent et un vaste outil de communication informatique graphique, précis, et efficace pour les équipes d'études, les équipes systèmes et les équipes de développement logiciel.

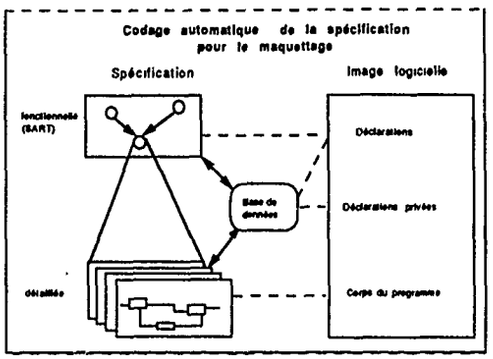
2.2 Maquettage

Les spécifications, aux différents niveaux d'abstraction, ne sauraient être finalisées sans la possibilité d'évaluer leur comportement dynamique très tôt dans le cycle de développement. Il est nécessaire d'introduire dans le cycle de développement des mini-cycles de validation, associés aux phases qui le composent, permettant de s'assurer que les spécifications issues d'une phase répondent aux besoins pour lesquels elles ont été établies. Ces opérations de validation s'appuient sur le maquettage, pour lequel VISA assure la traduction automatique de la spécification dans un langage de simulation.

Ce maquettage, mené en parallèle à l'écriture des spécifications, nécessite :

- une représentation logicielle de la spécification, réalisée en grande partie grâce à des traducteurs automatiques, ce qui limite les risques d'erreurs,
- une modélisation de l'environnement de l'équipement ayant pour but de donner un comportement réaliste aux entrées/sorties des fonctions spécifiées,
- un environnement de conduite de simulation et d'analyse de résultats.

Un effort particulier est porté sur la présentation des résultats des simulations, ceux-ci étant insérables dans le document d'ensemble de la spécification. La gestion de configuration, plus souple pour le maquettage que pour la spécification, est néanmoins sous-jacente et apporte à la phase de maquettage une démarche rigoureuse et précise (une simulation est réalisée à partir d'un certain état de la spécification, dans un contexte précis etc...; la gestion de configuration permet d'en retrouver et d'identifier le domaine de couverture théorique).



Le code image de la spécification est élaboré à l'aide, principalement, de traducteurs automatiques développés à SEXTANT Avionique :

- SART/ADA pour la spécification globale,
- GALA pour la spécification détaillée dont chaque élément représente un élément terminal de la représentation fonctionnelle (une PSPEC dans le formalisme SART),
- un générateur de code exploitant les informations des BD interne et externe, pour les déclarations d'entrées/sorties.

La conduite de simulation est conviviale:
- interface graphique (visualisation, poste de commande...),
- interactivité (lancement/arrêt des essais, définition des enregistrements/stimuli, modifications des paramètres internes de la spécification).

Le dialogue entre les fonctions à tester (l'image de la spécification) et leur environnement se fait par envoi/réception de messages inter-processus, ce qui permet de faire communiquer des langages différents tels que C, Le-Lisp, Fortran et ADA.

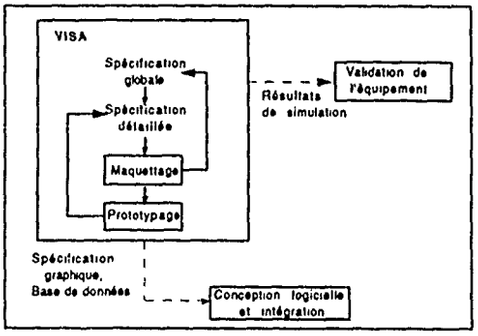
2.3 Prototypage

Ce terme désigne ici une opération qui relève de la même démarche que le maquettage avec la différence que l'animation des fonctions est effectuée en temps réel. Ceci se justifie pour les fonctions dont la validation nécessite la présence d'éléments réels ;

- certaines fonctions font intervenir le pilote et ne peuvent être validées qu'en présence d'un opérateur réagissant en temps réel :
- de même, il peut être nécessaire de valider une fonction dialoguant avec une autre en la plaçant dans le

contexte réaliste créé par la présence d'un équipement réel supportant cette autre fonction.

En phase de prototypage, VISA intègre les contraintes temps-réel du logiciel. Ces contraintes sont du ressort de la réalisation logicielle, mais sont prises en compte lors de la spécification détaillée. Ceci souligne la nécessité de considérer le projet dans son ensemble en liant étroitement les différentes phases les unes aux autres, tout en procédant à un affinage progressif: le maquettage ayant validé les contraintes amont (cahier des charges), le prototypage prévoit le comportement de la spécification soumise aux contraintes aval ou à l'environnement réel. Ces contraintes sont modélisées pour cette tâche.
Nous verrons au §4 que la validation quantitative s'appuie avantageusement sur des résultats de simulations, générés par l'outil de prototypage.



Les besoins pour le prototypage sont du même ordre que pour le maquettage : disposer d'outils pour produire, à partir d'une spécification globale, un code exécutable image de la fonction spécifiée, et pour mettre en oeuvre ce code dans un contexte temps réel. Ces outils utilisent les ressources du Banc de Validation (cf §4).

3. Conception et réalisation du logiciel

Les résultats des activités de spécifications, maquettage et prototypage sont exploités dans les phases de conception et de réalisation.

La conception et le schéma global dans lequel se place la structuration logicielle des équipements de pilotage-guidage sont parfaitement maîtrisés. Ces équipements entrent dans la catégorie des systèmes réactifs qui réagissent continûment à leurs entrées pour recalculer cycliquement leurs sorties. La phase de conception de ces équipements peut se schématiser en une répartition des éléments de la spécification détaillée dans des modules logiciels en fonction de contraintes d'implantation du logiciel.

Dans les équipements de pilotage et guidage, une faible partie du volume de code est constituée de logiciel commun réutilisé sur d'autres calculateurs. C'est le cas du moniteur temps réel, d'une bibliothèque de fonctions avioniques de base et d'une partie des tests de sécurité et de maintenance intégrée. La plus grande partie du logiciel est très spécifique d'un calculateur donné. Dans le cas du pilote automatique de l'Airbus A320, la répartition en volume des fonctions spécifiques du pilotage automatique est la suivante :

- 45% pour les calculs de logique et lois de pilotage,
- 30% pour la gestion des entrée/sorties,
- 20% pour les séquencements,
- 5% pour diverses fonctionnalités.

La part de logiciel est de plus en plus importante dans les équipements, mais les délais impartis à la réalisation restent constants. L'utilisation d'outils de génération automatique de code est donc un moyen d'augmenter la productivité tout en maîtrisant la qualité du logiciel produit.

Il apparait judicieux de générer automatiquement le code des fonctionnalités représentant le plus fort pourcentage de volume de code sujet à de fréquentes modifications. Ainsi GALA est un outil générant automatiquement le code du calcul de la logique et des lois de pilotage, tandis que GALI génère le code de la gestion des entrées/sorties.

Les outils de génération de code GALA et GALI développés par SEXTANT Avionique sont fondés sur l'approche suivante :

La génération automatique du code des modules logiciels, nécessite la formalisation d'un langage de spécification détaillée. La génération automatique d'un module logiciel est alors la transcription exacte de la spécification de ce module, décrit à l'aide de ce langage, en code source acceptable par un compilateur.

3.1 GALA : Génération Automatique de Logiciel Avionique

GALA (Génération Automatique de Logiciel d'Avionique) génère automatiquement le code source exécutable des modules logi-

ciels spécifiés sous forme graphique ainsi que la documentation associée.

3.1.1 Le langage

Le même langage est utilisé pour la spécification détaillée sous VISA et pour la programmation par GALA.

Il s'apparente aux langages à flot de données synchrone, avec lesquels un système est représenté par un réseau d'opérateurs connectés par des liaisons.
Un opérateur (un symbole graphique) représente une fonction. Une liaison (un fil) représente une donnée.

Ce formalisme présentent les avantages suivants :

- Il est particulièrement bien adapté à la description des automatismes et de la logique opérationnelle,
- La syntaxe est simple et bien définie,
- Les contraintes de séquencement dans l'exécution d'un programme résultent uniquement des dépendances fonctionnelles entre les variables,
- Les temps de réaction des opérateurs du réseau sont supposés négligeables par rapport aux cadences de circulation des données (cadences définies par le cycle d'activation). Cette hypothèse permet de faire abstraction des contraintes temporelles.

Chaque symbole graphique correspond à une fonction décrite par un algorithme validé et certifié et à un sous-programme codé dans le langage cible. L'ensemble des fonctions est regroupé dans une bibliothèque.
Cette bibliothèque est extensible. Quand un projet identifie un nouveau sous-programme utilisé de façon répétitive, il suffit de créer un symbole graphique avec la caractérisation de ses broches de connexion, puis de le valider et de l'installer dans la bibliothèque.

Les informations complémentaires nécessaires au codage sont prises en compte par l'intermédiaire de paramètres associés aux symboles.

3.1.2 L'utilisation

Le processus de production de logiciel par GALA se déroule en plusieurs phases :

- Edition des diagrammes. Dans le cadre de l'atelier, cette édition est effectuée pendant la phase de spécification détaillée (VISA).

- Analyse et contrôle de cohérence du diagramme en accord avec un certain nombre de règles syntaxiques (par exemple, toute broche de symbole doit être connectée) et sémantiques (par exemple, si un chemin de données comporte une boucle, un opérateur de mémorisation doit être obligatoirement présent sur cette boucle). Une partie de ces contrôles est effectuée sous VISA en phase de spécification.

- Génération automatique de code en deux étapes : d'abord la génération de code symbolique indépendant du langage, suivi de la traduction de ce code symbolique dans le langage de programmation cible.

L'utilisation d'un nouveau langage de programmation cible pour un projet donné implique seulement le changement du traducteur final. Les traducteurs disponibles actuellement sont : PASCAL, PLM, FORTRAN et ADA.

3.2 GALI : Génération Automatique de Logiciel d'Interface

Le langage de spécification utilisé par GALI est un langage textuel propre aux traitements des entrées/sorties.

GALI s'appuie sur une base de données contenant les caractéristiques de tous les signaux en entrée et en sortie d'un équipement. Cette base de données est renseignée en phase de spécification.
Le concepteur enrichit la définition des signaux en y apportant des informations complémentaire (des contraintes logicielles par exemple). GALI assure alors la définition et la mise à jour du découpage fonctionnel en relation étroite avec l'outil de gestion de configuration, élabore la documentation d'interface de l'équipement et les spécifications de codage dans le langage approprié .

GALI génère le code des modules de traitements d'entrées sorties en conformité avec leur spécification de codage ainsi que les modules de déclaration des données afin d'en assurer la cohérence avec leur environnement. GALI utilise les mêmes traducteurs que GALA.

3.3 Bilan d'utilisation

Les avantages apportés par la génération automatique de code sont importants :

Les outils de codage automatique GALA et GALI transcrivent fidèlement la spécification de codage en langage de programma-

tion. Aucune intervention humaine n'est nécessaire à ce stade.
Tout risque d'introduction de défaut de codage est supprimé, ce qui permet d'obtenir beaucoup plus rapidement le niveau requis de qualité du logiciel. Les tests structurels (tests unitaires "boite blanche") des modules générés automatiquement peuvent être supprimés.

Ces outils utilisent directement les informations issues de la spécification et apportent une facilité dans la traçabilité de ces informations ainsi qu'une bonne cohérence sur l'ensemble de l'atelier de développement, dans lequel chaque information n'est définie qu'une seule fois, évitant les risques d'incohérence d'une double définition. La cohérence parfaite entre documents de spécification et code exécutable et la concentration du développeur sur les tâches de spécification sont autant de facteurs supplémentaires de qualité du logiciel produit.

La suppression de la phase de tests structurels (tests "boite blanche") des modules générés automatiquement n'est autorisée par les autorités de certification que dans la mesure où les outils ont été qualifiés, ce qui est le cas pour les programmes Airbus A320 et A340.

4. Intégration et validation des équipements

4.1 Méthodes

Cette partie traite de l'ensemble des essais fonctionnels réalisés après intégration du matériel et du logiciel.
Ces essais font suite aux tests réalisés par les équipes de développement matériel et logiciel.

L'équipe matérielle aura intégré les différents éléments du calculateur, vérifié les signaux d'alimentation, les différentes fonctions des cartes, le câblage de fond de panier, effectué les auto-tests.
L'équipe logicielle aura validé le moniteur temps réel, la bibliothèque avionique GALA. Les tests unitaires auront été effectués pour les modules codés manuellement.

L'objectif de ces essais fonctionnels est de vérifier :

- l'intégrité du système, c'est à dire que toutes les fonctions prévues pour obtenir la sûreté de fonctionnement sont correctement implantées,
- la conformité de l'équipement réalisé par rapport aux différents documents de spécification (interface, lois de pilotage, logique opérationnelle)

Les opérations d'intégration/validation s'organisent en quatre phases :

Phase 1 : Intégration et Validation par processeur

Les objectifs de cette phase sont de vérifier:

- l'intégration hard/soft processeur par processeur sous l'aspect intégrité (tests de sécurité, auto-tests)
- les dialogues inter-processeurs.

Les outils utilisés sont les bancs de test de l'équipement et les moyens d'investigation associés, tels qu'analyseur logique et émulateur.

Phase 2 : Tests en boucle ouverte (l avion n'est pas dans la boucle)

Les tests boucle ouverte s'effectuent sur un calculateur complet et fermé (tests boîte noire).
Dans cette phase de test, les sorties de l'équipement (ordres gouvernes et manettes des gaz) ne sont pas envoyées à l'environnement simulé.
Les objectifs de cette phase sont de vérifier :

- le comportement dynamique du système sous les aspects architecture générale et séquencement temps réel,
- les flots de données à l'intérieur du système,
- globalement le monitoring.

Les tests boucle ouverte sont répartis en deux sous-phases :

a) Tests de la logique

La logique comprend à la fois la logique opérationnelle liée au contrôle du vol par l'équipage et la surveillance et reconfiguration des entrées/sorties de l'équipement.
Sont testés dans cette phase les engagements des modes et des sous-modes au moyen des postes de commande de l'équipement ainsi que les affichages des différentes informations

b) Tests des lois de pilotage

Le but de cette phase de test est une validation dynamique et quantitative de la fonction, par exemple : une tenue de cap, d'altitude ou d'ILS.

Le calculateur reçoit un ensemble de signaux cohérents correspondant à une configuration avion donnée sur lesquels se superposent un ou plusieurs stimuli.
Le principe retenu est un recoupement quantitatif par rapport à des simulations effectuées sur un logiciel image implanté sur un ordinateur sol (mainframe ou station de travail).
Des sollicitations du type échelon, créneau ou rampe sont appliquées sur les entrées capteurs, à la fois sur l'équipement réel et la fonction simulée.

La mise en oeuvre est effectuée sur le Banc de Validation Avionique (BVA).

Phase 3 : Tests en boucle fermée (l'avion est dans la boucle)

L'objectif de cette phase est de tester le comportement en dynamique de l'équipement dans les phases réelles de vol.
Dans cette phase, le calculateur reçoit les réponses de l'avion aux commandes générées par lui-même au travers d'une modélisation fine de l'environnement (avion, moteur, capteurs, autres équipements en relation avec l'équipement en test).
Le principe retenu est un recoupement quantitatif par rapport à des simulations effectuées sur un logiciel image implanté sur un ordinateur sol.
Les stimuli sont maintenant issus des postes de commande avion du pilote automatique tels que le pilote peut les envoyer "en situation" dans des conditions opérationnelles de vol.
Tous les modes sont ainsi testés : modes de croisière ou d'atterrissage automatique ainsi que la logique d'enchaînement des modes et de leurs différents états (modes armés, actifs,etc,...)
Dans cette phase l'ensemble de la fourniture est présente sur le Banc de Validation Avionique : postes de commande, afficheurs de mode, configuration multi-calculateurs.

Phase 4 : Tests de non-régression des états successifs du système

Des tests de non régression des états successifs de l'équipement sont effectués sur des scénarii de vol complets. Les réponses de l'état n+1 sont comparées aux réponses de l'état n pour les fonctions non modifiées entre les deux états. La mise en oeuvre est effectuée de façon automatisée sur le banc de Validation Avionique (BVA).

4.2 Evolution des méthodes de validation

Une évolution des méthodes orientée vers l'automatisation du passage des procédures de test est en cours.
En effet, le passage des tests réclame un ensemble d'actions de l'opérateur sur le banc :

- configuration de l'environnement de simulation : choix de la version de simulation, de ses conditions d'initialisation,
- préparation du calculateur à tester : reset, auto-tests,
- actions sur les postes de commande avion : engagement du calculateur, engagement de modes, sélection de consignes...,
- commandes sur les moyens d'espionnage et de sollicitation du banc.

Deux difficultés sont identifiées dans ce processus manuel :

a) Le déterminisme

Lors d'un test effectué manuellement, les actions de l'opérateur interviennent à des instants aléatoires.
Or certains tests (tests de recoupement avec une simulation effectuée sur un logiciel image, tests de non régression) nécessitent de respecter un chronogramme précis d'enchaînement des actions.

b) La charge de travail des machines et des hommes

La complexité croissante des fonctions des calculateurs provoque une augmentation du volume des tests de validation. Il devient nécessaire de libérer l'opérateur des tâches répétitives afin qu'il puisse se concentrer sur le contrôle des résultats.

Ces deux difficultés mettent en évidence la nécessité de pouvoir, par un automate, "simuler" l'opérateur, ou tout au moins celles de ses actions :

- qui doivent être exécutées plus rapidement que ne le permet le temps de réaction d'un opérateur,
- qui ne nécessitent pas un niveau d'expertise important.

L'outil en cours de mise au point répondra aux deux objectifs :

- dérouler automatiquement des séquences d'actions déterministes,
- exécuter des tests en "batch".

Ces deux niveaux d'utilisation de l'automate sont basés sur des mécanismes qui s'appuient sur la structure existante du banc par ajout d'une couche "simulation de l'opérateur" au dessus des entrées "classiques" de l'outil.

L'automate pourra agir sur tout le Banc, constitué de matériels et de logiciels hétérogènes : stations de travail, stations graphiques, ordinateurs temps-réel, moyens de tracé, boîtiers de coupure...

4.3 L'outil : le Banc de Validation Avionique (BVA)

Le BVA est un ensemble de matériels et de logiciels permettant de tester les logiciels des calculateurs embarqués. Il est utilisé dans tous les programmes civils et militaires majeurs dans lesquels SEXTANT fournit des équipements de conduite du vol : Airbus A310, A320, A340, hélicoptère franco-allemand TIGRE, Mirage 2000, C135FR, puis RAFALE. pour le CET (Calculateur d'Elaboration de Trajectoires).

Architecture fonctionnelle

Un banc est constitué :

- d'une simulation temps réel sur calculateur de type GOULD/ENCORE comprenant :
  - un moniteur de simulation chargé de la coordination d'ensemble,
  - les modèles avion, moteur, capteurs,
  - une interface temps réel chargée des échanges avec les autres constituants du banc.
- d'une interface opérateur sur station de travail en frontal du calculateur principal supportant tout le dialogue de conduite de simulation et de mise en œuvre des moyens d'essai avec fonctions de stimulation, d'espionnage, et de sélection des enregistrements,
- d'instruments simulés sur station graphique de type Silicon Graphics (en l'absence de l'environnement réel),
- d'un pupitre dans lequel viennent s'insérer :
  - une "planche de bord" avec les éléments réels du poste de pilotage (postes de commande PA, visualisations électroniques)
  - des commandes cockpit (becs, volets, train, aérofreins, ...),
  - des espionneurs d'entrées-sorties,
  - des traceurs rapides,
  - des tiroirs de coupure d'alimentation,
  - les supports calculateurs.
- d'une chaîne d'entrées-sorties permettant le couplage de la simulation temps réel avec le ou les équipements à valider d'une part et le pupitre d'autre part.

Le banc est relié à un poste de tracé et d'aide à l'analyse des résultats d'essai.

Evolutions vers un Banc pour systèmes intégrés

Deux axes d'évolution sont envisagés :

- élargir le nombre d'équipements qu'il peut accueillir pour valider des systèmes intégrés d'avionique comprenant par exemple des capteurs et des visualisations,
- avoir la capacité de remplacer progressivement en cours d'intégration les éléments prototypés par les éléments réels.

5. Outil de gestion de configuration

PALAS : Production Assistée de Logiciel d'Application Structuré

Le développement du logiciel d'un équipement de pilotage-guidage est confronté à des contraintes importantes :

- le cycle de développement est court pour tenir des délais de plus en plus réduits,
- les fonctionnalités à intégrer amènent à embarquer un volume croissant de logiciel,
- le cycle de vie de l'équipement est long et l'environnement de développement est susceptible d'évoluer,
- la sécurité et la fiabilité du développement sont prépondérants,
- la nécessité de faire vivre plusieurs variantes d'un même logiciel, au même rythme et en parallèle.

La prise en compte de ces contraintes rend nécessaire l'utilisation d'une structure d'accueil facilitant la gestion des évolutions des projets et l'intégration des divers outils logiciels utilisés, tout au long des phases de spécification, conception, codage, tests unitaires, intégration, validation et maintenance.

PALAS répond à l'ensemble de ces besoins. C'est une structure d'accueil de haut niveau qui prend en compte l'ensemble des dimensions d'un développement logiciel.

Les principales fonctions offertes par PALAS sont :

- la structuration d'un produit logiciel en arborescence avec liens,
- le contrôle des accès et interface utilisateur,

- la construction et le tests des configurations,
- la mise en place de versions officielles,
- le développement de versions de logiciel en parallèle,
- le contrôle complet et l'enregistrement des modifications,
- la gestion des historiques de tous les composants logiciels,
- l'intégration des outils du projet.

5.1 Organisation du projet

PALAS permet d'une part d'organiser le projet et de connaître à tout instant l'état d'évolution de celui-ci, d'autre part de définir les tâches à réaliser par chaque membre de l'équipe de développement.
PALAS permet également de synchroniser des développements dans le cas d'un projet partitionné en plusieurs sous-projets.
Chaque utilisateur PALAS travaille dans un "espace privé" qui lui est propre, dans le cadre d'une tâche préalablement définie par le responsable projet.

Le responsable de projet peut définir des autorisations d'accès à chaque membre d'un projet. Le rôle de chacun est alors bien défini, ce qui induit :

- une organisation claire des équipes de développement,
- une définition claire et complète, pour chaque membre de l'équipe, de son contexte de travail, ainsi que des informations qui lui sont nécessaires,
- le masquage de certaines parties de développement (par exemple dans le cadre d'une coopération ou d'une sous-traitance).

Chaque "entité" définie avec PALAS possède un nombre variable "d'objets" (de fichiers) associés, principalement identifiés par les outils s'appliquant sur l'entité.

Une entité PALAS peut donc correspondre à un modèle complet de spécification, à un module logiciel (avec l'ensemble de ces fichiers de documentation, code source, code objet exécutable...), à un jeu de tests, etc,... Ce concept permet de gérer les composants produits lors des phases du cycle de vie du logiciel.

5.2 Gestion de configuration

La description et l'exploitation des "liens" entre les différentes entités d'un projet permettent de mettre en oeuvre des mécanismes de configurations complètes ou partielles de versions du logiciel et le maintien d'intégrité entre plusieurs documents, par exemple entre la spécification, un module logiciel et le résultat des tests.
PALAS réalise entièrement et de façon automatique toutes les tâches de gestion de configuration et permet donc de maintenir la cohérence entre l'ensemble des entités d'un projet, depuis l'expression des besoins jusqu'à la mise en exploitation, pour laquelle il dispose d'un mécanisme de mise en place de versions officielles.

5.3 Développement en parallèle

Il est fréquemment nécessaire de faire vivre simultanément plusieurs variantes d'un même logiciel. Ces variantes peuvent être liées à des configurations système différentes ou à différents niveaux d'intégration. Il est alors important de conserver la plus grande part de composants communs entre les diverses variantes.

PALAS offre toutes les commandes pour développer en parallèle et continuer le développement en parallèle ou le faire reconverger. Il autorise la gestion simultanée du plusieurs configurations d'un même sous-ensemble du logiciel, tout en préservant des divergences les composants communs à ces différentes versions.

5.4 Suivi des modifications

PALAS offre la possibilité d'établir des procédures rigoureuses pour les modifications par l'intermédiaire des Décisions d'Intervention logicielle (DIL). Les décisions d'interventions logicielles, qui résultent d'une analyse d'évolution, sont décrites sous PALAS en précisant les entités touchées par chaque évolution, et regroupées au sein de Rapports de Modification (RM) qui rythment l'évolution du logiciel d'un état stable à un autre.

PALAS réalise le suivi des évolutions et gère l'historique de chaque composant du projet ce qui permet d'en maîtriser l'état à tout instant du cycle de vie et de garantir ainsi la sécurité et la fiabilité du développement.

5.5 L'intégration d'outils

PALAS offre un ensemble de services pour intégrer les divers outils utilisé sur un projet et contrôler le processus de développement à travers la mise en oeuvre de ces outils.

PALAS permet de réutiliser des procédures mise en place sur d'autres projets, voire des procédures standardisées au sein d'une entreprise.

L'intégration d'outils de développement logiciel se fait par l'intermédiaire de procédures de production ou "chaînes de production" qui peuvent aller du simple appel à un outil, jusqu'à l'enchaînement complexe de plusieurs outils.
PALAS permet de définir des chaînes de production fonctionnant dans des environnements spécifiques liés à l'environnement cible (les bancs de validation par exemple), ou à l'environnement d'un outil (les stations de travail pour VISA par exemple).

PALAS permet de prendre en compte les contraintes d'hétérogénéité de matériels de développement, et offre les mécanismes nécessaires pour maintenir l'intégrité du projet sans intervention manuelle de l'utilisateur.

PALAS est un outil industriel utilisé dans le cadre des grands programmes : A320, A340, HAP, C135FR, ARIANE IV.

6.Conclusion

L'atelier de développement de logiciels de pilotage-guidage s'appuie sur une méthodologie stable, et éprouvée par SEXTANT Avionique pour le développement d'un grand nombre d'équipements. L'atelier permet de mettre en oeuvre rigoureusement cette méthodologie, à travers l'emploi d'outils adaptés à chaque phase du cycle de développement. Il assure la communication des informations issues de ces outils pour couvrir l'ensemble de la méthodologie sur tout le cycle de vie du projet.
L'atelier est donc le garant d'un bon niveau de qualité des équipements réalisés.

ATELIER DE SPÉCIFICATION / MAQUETTAGE POUR LES SYSTEMES DE GESTION DU VOL

Hugues ROBIN, Jean-Christophe MIELNIK
SEXTANT Avionique
Division Conduite du Vol
Aérodrome de Villacoublay
78141 Vélizy Cedex
FRANCE

1. Introduction

Les systèmes embarqués développés par la Division Conduite du Vol de SEXTANT Avionique évoluent sans cesse au niveau de leurs fonctionnalités, des moyens de développement et des techniques d'implémentation.

L'évolution des fonctionnalités peut être observée sur deux plans :

- l'automatisation de la conduite du vol correspond à l'évolution du concept de Pilote Automatique vers celui de Système de Gestion du Vol : ces systèmes mettent à la disposition des pilotes un ensemble de fonctions d'aide au pilotage, depuis l'automatisation de la traditionnelle tenue de consigne jusqu'à l'aide à la reconfiguration du plan de vol par le pilote en cours de mission, avec prise en compte de la préparation de mission au sol,
- de plus, l'intégration de nouvelles fonctionnalités est rendue possible par la communication avec les systèmes de détection et de contrôle (radars), ces systèmes pouvant être terrestres ou aériens, statiques ou mobiles.

Au niveau des moyens de développement, le logiciel joue un rôle de plus en plus important. Il complète les techniques de l'automatique principalement utilisées pour les systèmes de pilotage et guidage.

Si, aujourd'hui, les langages de programmation de type ADA ainsi que les méthodes de spécification et de conception sont couramment mis en oeuvre, les technologies du logiciel ne cessent d'évoluer : les langages orientés objets et les systèmes experts actuellement utilisés en phase d'étude feront bientôt partie des technologies embarquées.

Le rythme important avec lequel ces évolutions fonctionnelles et techniques sont menées nécessite de mettre en place des moyens pour maîtriser et anticiper ces évolutions. Au niveau de la définition des fonctionnalités, ceci se traduit par un renforcement du dialogue que SEXTANT entretient avec ses différents interlocuteurs : les compagnies aériennes, les pilotes et les avionneurs. Des méthodes accompagnées d'outils supports sont mises en place pour formaliser ces dialogues afin d'aboutir à des expressions de besoin complètes et non ambiguës.

Le département Avant-Projets de la Division Conduite du Vol est responsable de la définition des moyens de spécification fonctionnelle utilisés dans les phases amont du cycle de développement : ces phases sont d'autant plus importantes qu'une erreur introduite à ce stade est amplifiée au cours des phases aval et que sa découverte tardive entraîne des retours en arrière toujours très coûteux.

Ce document présente les différents besoins qui apparaissent dans ces phases amont ainsi que les divers environnements mis en place à SEXTANT Avionique pour la spécification et le maquettage des logiciels de gestion du vol.

2. Les besoins en moyens de spécification

Les besoins sont issus du constat d'une contradiction concernant les documents de spécification :

- ils constituent les premiers documents contractuels d'un projet et figurent de ce fait parmi les documents les plus importants,
- leur forme, des documents papier composés d'un grand nombre de pages avec des références croisées non structurées, rend problématique voire parfois impossible leur exploitation.

Ainsi, les outils d'aide à la phase de spécification doivent non seulement permettre d'exprimer aisément les exigences fonctionnelles, mais aussi de vérifier rapidement et à posteriori la validité des documents de spécification par rapport à ces mêmes exigences.

La solution consiste à mettre en oeuvre des moyens de spécification formelle : leur caractère formel, par opposition au caractère informel inhérent aux langages naturels souvent sujets à interprétation, évite les ambiguïtés et les incomplétudes.

Ceci est parfois obtenu au détriment de la lisibilité des spécifications. Une solution complète consiste alors à associer une activité de maquettage à l'utilisation de méthodes de spécification formelle. Il devient ainsi possible de valider très tôt dans le cycle de développement l'expression des besoins, la phase intégrée de spécification/maquettage apportant le double avantage :

- de mettre à la disposition des clients un document sans ambiguïté ni incomplétude et des moyens permettant une exploitation rapide et lisible,
- de mettre à la disposition des réalisateurs, une expression des besoins claire et complète pour les fonctions à réaliser : le caractère formel des spécifications permet de mettre en place les moyens d'assurer la traçabilité avec la phase de conception logicielle et, lorsque cela est possible, avec la phase de génération automatique de code embarqué [13].

2.1.Intégration de la spécification et du maquettage au sein d'un même environnement

SEXTANT Avionique, considérant spécification et maquettage comme indissociables, a mis en place plusieurs environnements, chacun basé sur une méthode de spécification particulière à laquelle est associé un générateur de code. La simulation obtenue par l'exécution du code généré se fait dans le cadre d'un environnement de maquettage pré-existant. Les fonctionnalités communes à ces environnements de spécification/maquettage sont décrites dans la suite de ce document.

2.1.1.Spécification et "vérifications statiques"

A une méthode de spécification formelle, sont associées deux notations strictement équivalentes : la première est graphique et forme l'interface dont le spécifieur dispose pour exprimer ses besoins. La seconde est textuelle et constitue le langage intermédiaire à partir duquel les vérifications liées à la méthode sont faites.

Ces vérifications sont au nombre de deux : la première, purement syntaxique, vérifie que tous les symboles utilisés par le spécifieur pour sa description formelle sont autorisés et que leur agencement les uns avec les autres correspond à la grammaire du langage. Cette vérification est facilitée lorsqu'à partir de la saisie effectuée à l'aide d'un éditeur graphique dédié, la représentation équivalente dans le langage intermédiaire est automatiquement générée.

La seconde vérifie qu'il n'y a pas d'incohérence au niveau de la sémantique de la méthode : par exemple, des cas de division par zéro et les conflits de types peuvent être détectés lors d'une phase d'analyse sémantique. La mise en évidence de ces incohérences, dont l'implémentation peut être très complexe, est faite de manière dynamique, c'est à dire lors de l'exécution du code maquette.

2.1.2.Maquettage et "vérifications dynamiques"

2.1.2.1.Mise au point de la spécification

L'animation ou la simulation d'une description formelle des spécifications a pour but de les valider. Cette phase de validation consiste à :

- vérifier que la description est non ambiguë et complète : c'est ce qu'on appelle "le debuggage de la spécification" qui consiste à détecter les erreurs dues au non respect de la méthode et qui n'ont pas pu être découvertes lors des vérifications statiques,
- vérifier que la description formelle exprime bien les besoins et que tous les besoins y sont exprimés : c'est ce qu'on appelle la "validation de la spécification".

Dans les deux cas, le spécifieur est amené à modifier la description formelle si l'exécution du code maquette associé ne correspond pas à ce qu'il attend, que ce soit du point de vue "debuggage" ou du point de vue "validation".

Pour faciliter la détection des erreurs, différents modes d'exécution du code maquette sont implémentés : plusieurs modes "pas à pas" (avec différentes valeurs du pas) et un mode en continu avec possibilité d'arrêt sur événement.

Pour faciliter la correction des erreurs dans la description formelle, la traçabilité est assurée entre les éléments de spécification et le code maquette.

2.1.2.2.Interface de simulation

L'interface de simulation constitue le moyen de dialogue entre l'environnement de spécification/maquettage et le spécifieur.

Outre les moyens d'édition nécessaires à la formulation des exigences fonctionnelles, le spécifieur dispose de moyens lui permettant de contrôler sa simulation:

- en envoyant aussi bien des stimuli (Run, Stop...) que des données,
- en visualisant des informations au cours d'une simulation.

Ces moyens sont basés sur les ressources qu'offrent aujourd'hui les stations de travail, en particulier l'utilisation de la souris, du multi-fenêtrage... (Cf. annexe 3).

La caractéristique principale de l'interface de simulation est d'être complètement dissociée de la description formelle. Pour chaque description on peut définir plusieurs interfaces possibles grâce à un langage de définition d'interface.

2.2.Intégration de plusieurs environnements de spécification/maquettage au sein d'un atelier

2.2.1.Position du problème

La multiplication d'environnements dédiés à des métiers très spécialisés, tous utilisés pour le développement d'un même système de Gestion du Vol, nécessite d'intégrer ces environnements entre eux.

Un environnement donné, constitué d'une méthode de spécification et d'un environnement de maquettage, est
- soit dédié à un métier particulier parmi lesquels on peut citer celui des lois de pilotage, de la logique du dialogue homme-machine,
- soit à vocation plus générale.

Chacun des métiers mis en oeuvre pour le développement des fonctions de Gestion du Vol doit faire face à une double évolution:
- du point de vue des moyens de spécification associés, comme par exemple l'apparition de nouvelles méthodes, les adaptations à apporter à l'interface homme-machine des outils supports,
- et du point de vue des domaines d'intervention des métiers qui sont de plus en plus nombreux au fur et à mesure de l'apparition de nouvelles fonctionnalités dans les fonctions de Gestion du Vol.

D'autre part, ces nouvelles fonctionnalités génèrent de nouveaux métiers et nécessitent un environnement de spécification/maquettage adapté : tel est le cas pour l'ELS dont la maîtrise demande la mise en oeuvre de techniques de type "Hypertext" non utilisées jusqu'alors pour le développement des systèmes embarqués.

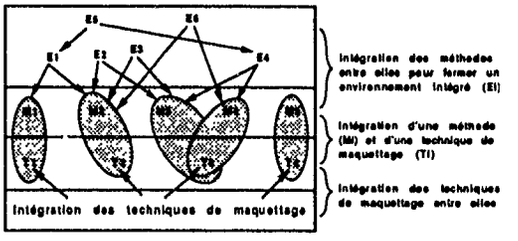
Des environnements de spécification/maquettage à vocation "généraliste" peuvent aider à la formalisation d'un métier particulier : ainsi, l'environnement OOA/KEE a permis de développer un environnement de spécification/maquettage d'une application ELS (cf. troisième partie de ce document).

2.2.2.Structure d'accueil

Les problèmes d'intégration se posent à plusieurs niveaux :
- intégration à un environnement d'une nouvelle méthode de spécification,
- intégration à un environnement d'une nouvelle technique de maquettage,
- intégration de plusieurs environnements de spécification/maquettage les uns avec les autres : pour des mêmes environnements, l'intégration peut être faite de différentes manières, en fonction du mode de fonctionnement du projet, de la répartition des tâches entre les différents intervenants...

La solution consiste à disposer d'une structure d'accueil constituée de méthodes et d'outils permettant de spécifier les différents impératifs d'intégration.

Cette structure d'accueil doit disposer de trois types d'outils :
- outils de définition de l'intégration de plusieurs méthodes entre elles,
- outils de définition de l'intégration des techniques de maquettage entre elles,
- outils de définition de l'intégration d'une méthode avec une technique de maquettage.



- Structuration en couches de la structure d'accueil -

A partir de définitions écrites avec ces outils, la structure d'accueil implémente, sans aucune intervention humaine supplémentaire, un nouvel environnement intégré de spécification/maquettage adapté.

3. Réalisations

Deux environnements de spécification/maquettage, destinés à la formalisation des "fonctions nouvelles", ont été mis en place au sein du Département Avant-Projets de la Division Conduite du Vol : le premier est basé sur la méthode SART (Structured Analysis & Real Time) pour la spécification fonctionnelle et le langage ADA pour le maquettage. Le second, qui s'appuie sur les langages orientés objets, a servi à formaliser la fonction ELS (Electronic Library System) et a conduit à la mise en place d'un environnement de spécification/maquettage dédié à ce type d'application.

3.1.Un environnement "classique": SART/ADA

3.1.1.la méthode SART considérée

La méthode SART est particulièrement adaptée à la spécification fonctionnelle des applications temps-réel. L'outil STP (Software Through Pictures) [1], un des outils du marché supportant la méthode SART, est largement diffusé à SEXTANT pour la spécification des fonctions assurées par les systèmes de Conduite du Vol.

Le terme "SART" est générique et désigne en fait deux méthodes particulières, celle de "Ward & Mellor" [3] et celle de "Hatley" [4]. La méthode SART est semi-formelle car certaines combinaisons syntaxiques n'ont pas une sémantique suffisamment précise. La méthode considérée ici est une extension formelle à celle de "Ward & Mellor" : les ambiguités sémantiques ont été supprimées.

3.1.1.1.Le modèle des traitements : SA

Le principe de SART consiste à décrire les fonctions assurées par un logiciel sous la forme d'une décomposition hiérarchique descendante, suivant le principe d'Analyse Structurée de YOURDON/DEMARCO [2]. Chaque niveau de description est complet en lui-même, mais la précision croît avec la profondeur.

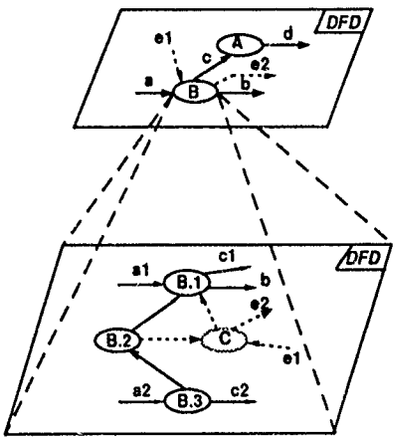
Une description SART est constituée d'une arborescence de diagrammes, la racine étant un diagramme de contexte et les autres noeuds des diagrammes de flux de données (DFD). Chaque diagramme contient des éléments de type : FONCTION, PSPEC, CSPEC, DATA-STORE, ENTITE-EXTERNE et CONNEXION. Seuls les éléments de type FONCTION se décomposent en un diagramme fils. Chacun de ces éléments possède des flux entrants et des flux sortants, ces flux assurant la connectique d'un modèle SART.

Deux types de flux sont définis : les flux de données et les flux de contrôles (ou événements). La différence se situe au niveau de la perception de ces flux par les éléments qui les reçoivent : les flux de contrôle sont perçus immédiatement par l'élément receveur car son comportement est susceptible d'être modifié immédiatement alors que les flux de données sont considérés comme de simples supports de données, données que l'élément receveur peut aller chercher de lui-même et quand il le décide.

3.1.1.2. Le modèle des contrôles : RT

Si le modèle des traitements décrit de manière statique le logiciel à réaliser, sous la forme de fonctions s'échangeant des données, le modèle des contrôles décrit la dynamique de ces fonctions les unes par rapport aux autres. Les diagrammes états/transitions associés à chaque CSPEC et les flux de contrôle entrant et sortant de ces CSPEC permettent de décrire cette logique.

Chaque DFD d'une spécification SART décrit un élément FONCTION d'un diagramme de niveau supérieur. La CSPEC est introduite sur un DFD pour décrire comment la fonction mère du DFD prend en compte les flux de contrôle qu'elle perçoit et comment elle émet les flux de contrôle sortants :



- Arborescence de DFD -

La CSPEC C permet de décrire les changements de comportement de la fonction B à partir non seulement des événements externes el et e2 mais aussi des événements internes (venant de B.2 ou émis vers B.1). Les changements de comportement de B

correspondent en fait à des changements de comportement des sous-fonctions B.1, B.2 et B.3. Via la CSPEC C, un diagramme états/transitions est associé à la fonction B. Il est tel que :

- l'ensemble des états correspond à un ensemble de configurations différentes des sous-fonctions B.1, B.2 et B.3,
- l'ensemble des événements conditions de transition est constitué par les flux de contrôle entrant dans la CSPEC : ces événements sont soit externes (el) soit internes (venant de B.2),
- l'ensemble des événements émis lors des transitions est constitué par les flux de contrôle sortant de la CSPEC : ces événements sont soit externes (e2) soit internes (émis vers B.1).

3.1.2.le traducteur SART/ADA

Les motivations qui ont conduit au choix de ADA comme langage cible sont au nombre de trois :

- les concepts du parallélisme mis en oeuvre dans ADA (notion de tâche, rendez-vous) ont permis d'implémenter un modèle SART comme un ensemble de processus en parallèle s'échangeant des données et se synchronisant.

- le concept de généricité permet de définir, pour chaque type d'élément de la méthode SART, un package générique que le générateur de code instancie pour chaque élément de chacun des diagrammes d'une spécification SART.

  Certains paramètres sont communs à tous les packages génériques (par exemple, la liste des flux sortants). D'autres ont des paramètres particuliers dépendant du type de l'élément SART (par exemple, le package correspondant à l'élément de type CSPEC a un paramètre "automate").

  En annexe 1, est jointe la partie spécification des six packages génériques. En particulier, pour le package correspondant aux éléments de type PSPEC, cinq tâches sont générées pour implémenter les deux types de flux : les flux de contrôle sont perçus immédiatement par la PSPEC alors que la consommation d'un flux de données est complètement désynchronisée de sa production. La communication entre ces cinq tâches, décrite dans le formalisme HOOD [7], est jointe en annexe 2.

- le choix de ADA comme langage de maquettage permet d'envisager une réutilisation partielle du code de maquettage dans la phase de codage : le code récupérable se situe au niveau de la description des PSPEC faite dans un pseudo-langage de type ADA.

3.1.3.l'environnement de simulation minimal

Dans un modèle SART, la description du dialogue entre la fonction spécifiée et son environnement est entièrement regroupée dans le diagramme de contexte : on y précise les flux de données et les flux de contrôles échangés entre les entités externes et la fonction que l'on spécifie.

L'idée de départ est de considérer que les entités externes sont elles-mêmes issues d'une spécification SART. Ceci permet de considérer l'environnement de maquettage comme pouvant être enrichi à chaque fois qu'une nouvelle fonction est spécifiée en SART, celle-ci devenant une entité externe utilisable pour la spécification d'une autre fonction.

Au départ, c'est à dire avant de disposer d'entités externes issues de spécifications SART, on dispose d'un environnement minimal de simulation constitué d'un ensemble d'entités externes de base à partir desquelles on peut en construire de nouvelles qui, elles, seront issues de manière automatique d'une spécification SART. Ces entités externes de base sont aussi bien de haut niveau et dépendent alors d'une application (par exemple "Guidage", "Pilote", "Paramètres avion", "Plan de vol" et "MCDU" pour la gestion du vol) que plus générales comme par exemple "clavier", "visu", "fichier" et destinées à plusieurs types d'application.

3.1.4.Environnement de simulation étendu

L'environnement de simulation, utilisé pour valider la spécification, propose deux modes d'exécution :

- un mode "pas à pas" où la valeur du pas correspond à l'émission d'un flux de contrôle par un des éléments de la spécification,
- et un mode en continu avec possibilité d'arrêt pour observer et inspecter l'état de la simulation.

L'état de la simulation correspond aux états des différents éléments du modèle SART :

- l'état des PSPEC et des FONCTIONS : actif ou inactif,
- l'état courant de chaque diagramme états/transitions,
- la production et/ou la consommation des flux de données,

- l'émission des flux de contrôle.

L'accès à toutes ces informations est possible grâce à une instrumentation du code généré par du code permettant de garder, à tout moment, la traçabilité entre la simulation et la spécification SART.

Une extension prévue à cet environnement est son intégration avec l'environnement de spécification/maquettage dédié aux lois de pilotage (VISA) : un type d'intégration possible consiste à décrire les PSPECS non plus directement par une procédure ADA mais avec le formalisme de la méthode de spécification des lois de pilotage : ceci est d'autant plus facile que VISA permet la génération de code ADA.

D'autres extensions sont à l'étude :
- prendre en compte les particularités de la méthode de HATLEY, en particulier les tables d'activation/désactivation et les tables de décision,
- enrichir le modèle des contrôles par des langages formels de type Statecharts [8] et HMS [9],
- et le couplage avec l'environnement à vocation "fonction nouvelle" OOA/KEE.

3.2.Un environnement "avancé" : OOA/KEE

L'approche fonctionnelle en phase de spécification consiste à décrire le système à réaliser en listant les fonctions qu'il doit assurer et les flux d'information qu'elles s'échangent. Cette approche fonctionnelle descendante est naturelle en phase de spécification et correspond aux habitudes des spécifieurs. L'approche objet, qui favorise la réutilisation et la fiabilité du logiciel, est une approche ascendante plutôt utilisée en phase de conception. Lorsque les approches fonctionnelle et objet sont retenues respectivement en phase de spécification et en phase de conception, un problème de transition apparaît entre ces deux phases.

Pour éviter cet inconvénient, SEXTANT Avionique a expérimenté l'utilisation de l'approche objet dès la phase de spécification dans le cadre de certains projets de conduite du vol. Les résultats obtenus s'avèrent prometteurs : les spécifieurs se sont très vite adaptés et ont été très enthousiastes. Un environnement de spécification/maquettage basé sur l'approche objet a donc été mis en place pour aider à la formalisation de certaines des nouvelles fonctions de gestion du vol.

3.2.1.La méthode OOA considérée

Une méthode de spécification décrit aussi bien l'aspect statique que dynamique d'un logiciel. La méthode considérée ici est inspirée des travaux menés par Coad et Yourdon [5] ainsi que par Schlaer et Mellor [10] pour définir une approche objet dans les phases amont du cycle de développement.

Les concepts de base mis en oeuvre dans ces méthodes sont ceux des Langages Orientés Objets (LOO) dont le plus illustre est SMALLTALK.

3.2.1.1.Aspects statiques

La stratégie de description des aspects statiques est issue de la méthode OOA (Object Oriented Analysis) [5]. La stratégie globale de modélisation a été simplifiée en ne retenant que quatre des principaux concepts de la méthode OOA : il s'agit des concepts de CLASSE (identique à celui des LOO [11]), de STRUCTURE D'ASSEMBLAGE, de GENERALISATION/SPECIALISATION et de LIEN D'INSTANCES.

3.2.1.1.1.Classe

Une classe est un type abstrait de donnée défini par une liste d'attributs et de services (ou méthodes) caractéristiques de ses instances :

```
classe  PLAN-DE-VOL  :
attributs : aéroport de départ
            aéroport d'arrivée
            liste ordonnée de points de
             passage

services  :  modifier
             insérer-point-de-passage
             (point_précédent,nouveau_point)
             activer
fin classe
```

-Définition de la classe des plans de vol-

L'instance PARIS-ATHENES de la classe PLAN-DE-VOL associe des valeurs particulières aux attributs :

```
instance  PARIS-ATHENES  classe-mere PLAN-
DE-VOL  :
        aéroport de départ
                    = PARIS
        aéroport d'arrivée
                    = ATHENES
liste ordonnée de points de passage
                    = (BERNE  MILAN  BARI)
```

fin instance

- Définition de l'instance PARIS-ATHENES -

Chaque instance est susceptible de rendre les services définis dans la classe mère aux autres instances. Par exemple, si le pilote sollicite le service "insérer-point-de-passage(MILAN,ROME)" au plan de vol PARIS-ATHENES, son attribut "liste ordonnée de points de passage" sera modifié de la façon suivante :

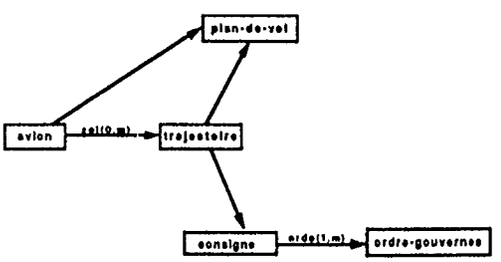
instance PARIS-ATHENES classe-mere PLAN-DE-VOL :
aéroport de départ
= PARIS
aéroport d'arrivée
= ATHENES
liste ordonnée de points de passage
= (BERNE MILAN ROME BARI)
fin instance

3.2.1.1.2.Lien d'instances

Le concept de lien d'instances (ou relation) s'inspire des modèles relationnels binaires de description des données (modèle entité-association en particulier) utilisés pour définir les schémas conceptuels des bases de données [12].

Un lien entre deux classes d'objets est unidirectionnel et met en relation une instance de la classe de départ avec une ou plusieurs instances de la classe d'arrivée, ce nombre étant défini par la cardinalité associée au lien par un couple de valeurs : (cardinalité minimum, cardinalité maximum).

Plusieurs types de relation existent :
- la relation "est composé de",
- la relation "est composé d'une collection de",
- la relation "est composé d'une liste ordonnée de",
- et les relations définies par le spécifieur.



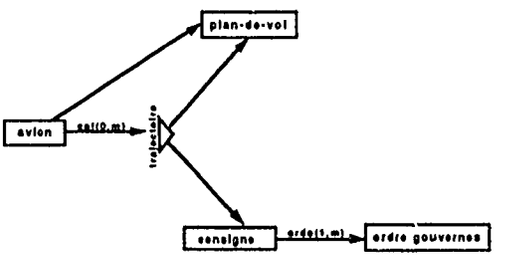
- Liens d'instances -

Sur le schéma, chaque instance de la classe AVION est composée d'une collection d'instances de la classe TRAJECTOIRE, cette collection peut être vide (la cardinalité minimum est nulle) ou contenir un nombre non borné d'instances (la cardinalité maximum est "m"). De la même manière, chaque instance de la classe CONSIGNE est composée d'une liste ordonnée d'instances de la classe ORDRE-GOUVERNES, cette liste doit contenir au moins une instance (la cardinalité minimum est égale à 1).

Chaque instance de la classe AVION est, à tout moment, en relation avec 0 ou 1 instance de la classe PLAN-DE-VOL, la relation "plan-de-vol-actif" a été définie par le spécifieur.

3.2.1.1.3.Structure d'assemblage

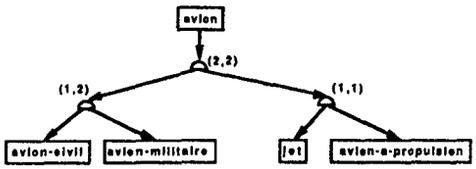
Le concept de structure d'assemblage permet de définir des classes d'objets comme des n-uplets d'autres classes : il se rapproche de la notion "d'enregistrement" que l'on trouve dans les langages de programmation structurée tels que PASCAL et ADA. Pour illustrer l'emploi de ce concept, supposons que la classe TRAJECTOIRE est entièrement définie par le simple fait que chaque instance est composée d'une CONSIGNE et d'un PLAN-DE-VOL. La classe TRAJECTOIRE ne dispose donc pas d'attributs et de services propres et la notation associée à la classe TRAJECTOIRE devient :



- Liens d'instances entre structure d'assemblage et classes -

3.2.1.1.4.Généralisation/spécialisation

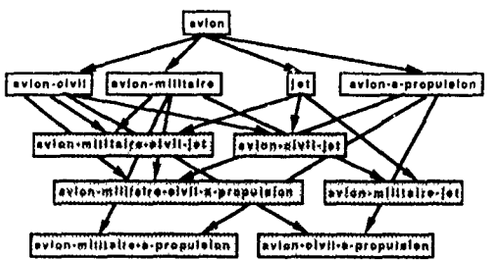
Ce concept est en fait une autre formalisation du concept d'héritage (simple et multiple) des LOO. Il a l'avantage de simplifier le graphe d'héritage en ne faisant apparaître que les classes élémentaires. Ce sont les cardinalités associées aux noeuds du graphe qui définissent les sous-classes obtenues par héritage des classes élémentaires et leur caractère d'instanciabilité (le fait qu'une classe peut ou ne peut pas avoir des instances dans le "monde réel", c'est à dire celui qui concerne le spécifieur) :



-Graphe de généralisation/spécialisation-

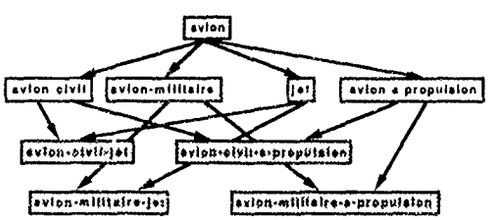
La lecture de ce graphe permet de déduire les assertions suivantes :

- Les classes "avion-militaire", "avion-civil", "jet" et "avion-à-propulsion" héritent des caractéristiques (attributs et services) de la classe "avion".
- Les cardinalités expriment les héritages multiples ainsi que les classes instanciables. Le graphe d'héritage équivalent dans le formalisme habituel serait le suivant (les classes instanciables apparaissent en grisé) :



- Graphe d'héritage équivalent au graphe de généralisation/spécialisation -

La cardinalité (1,2) permet de faire remonter au niveau du noeud père (celui qui porte la cardinalité (2,2)), les classes "avion-militaire", "avion-civil" et la classe composée "avion-militaire-civil". Si cette cardinalité avait été (1,1), les classes remontées se seraient limitées aux classes "avion-civil" et "avion-militaire" : un avion n'aurait pas pu avoir à la fois les caractéristiques d'un avion civil et d'un avion militaire. Le graphe équivalent dans le formalisme habituel aurait été le suivant :



3.2.1.2.Aspects dynamiques

La stratégie de description des aspects dynamiques est issue de la méthode OOSA (Object Oriented Systems Analysis) [10]. Il s'agit d'associer à chaque classe d'objets un diagramme états/transitions décrivant le "cycle de vie" des instances.

Ce diagramme décrit la logique d'envoi de messages entre objets en fonction des changements de valeur des attributs.

La sollicitation d'un service d'un objet receveur par un objet demandeur s'appelle "l'envoi de message". Deux types de sollicitations sont possibles : le premier est bloquant, l'objet demandeur se bloque jusqu'à ce que le service ait été complètement rendu (envoi de message synchrone), le second permet à l'objet demandeur de solliciter un service d'un autre objet sans attendre qu'il ait été rendu (envoi de message asynchrone).

La logique de déclenchement des services, décrite par les diagrammes états/transitions, est spécifiée au niveau des classes : les sous-classes et les instances héritent de cette dynamique et en particulier en cas d'héritage multiple, la sous-classe hérite de chacun des diagrammes des classes mères.

3.2.2.Le traducteur OOA/KEE

Le traducteur OOA/KEE a lui-même été écrit en KEE. KEE (Knowledge Engineering Environment) est un environnement de développement basé sur CommonLisp destiné aux applications mettant en oeuvre un système expert raisonnant sur une base de connaissance structurée grâce à une représentation objet. Outre les fonctionnalités d'un langage de frames, l'interface très conviviale (multi-fenêtrage, affichage de graphes...) a facilité le développement du traducteur.

Grâce à la similitude des concepts de l'OOA et de ceux des LOO, l'implémentation des aspects statiques de la méthode n'a pas posé de problèmes particulier : par exemple, la notion de "facette d'attribut" et en particulier celle de "valueclass", qui définit le(s) domaine(s) de valeur d'un attribut, a permis d'implémenter la notion de lien d'instances.

Pour les aspects dynamiques de la méthode, deux fonctionnalités de KEE/CommonLisp ont été avantageusement utilisées : il s'agit de la notion de "démon" (ou de "valeur active") et de la notion de parallélisme.

Un "démon" est une action associée à un attribut et qui est déclenchée à chaque accès à ce dernier (en lecture, en écriture ou les deux). Dans l'environnement KEE, un démon particulier, associé à chaque attribut d'un objet, évalue les conditions des transitions sortantes de l'état courant de l'objet : si la nouvelle valeur valide une condition de transition, l'action de transition associée est alors effectuée. La dynamique est ainsi entièrement dirigée par les changements de valeur des attributs.

L'existence du "parallélisme" est nécessaire pour implémenter la sollicitation de service non bloquante pour l'objet demandeur : les fonctionnalités de "multi-tasking" offertes par CommonLisp ont permis de réaliser une primitive d'envoi de message asynchrone "send(objet_destinataire,service)".

3.2.3.L'environnement de maquettage

De la même manière que l'environnement SART/ADA, l'environnement OOA/KEE offre au spécifieur

- des moyens de traçabilité du code maquette par rapport à la description OOA correspondante,
- différents modes d'exécution,
- un langage de définition de l'interface de simulation : il comporte un certain nombre de primitives, dont la primitive "DISPLAY" qui permet d'associer à un attribut une représentation graphique (jauge, thermomètre, compteur...) qui visualise sa valeur. Un autre primitive "DISPLAY_2D" permet de visualiser dans un plan l'évolution des valeurs de deux attributs.

Si, comme pour l'environnement SART/ADA, l'instrumentation du code généré a été une des techniques d'implémentation de l'environnement de maquettage, le calcul symbolique, fondement du langage Lisp, a accéléré le développement et réduit le code généré.

Une des fonctionnalités particulière induite par la méthode est le mécanisme d'instanciation. La méthode OOA définit des classes d'objet et les relations entre les futures instances de ces classes. Une simulation comporte une première phase d'initialisation qui consiste à créer les instances et à initialiser les relations qu'elles ont les unes avec les autres. C'est grâce aux cardinalités des relations que se fait la création des instances : la puissance du mécanisme d'instanciation permet, dès la phase d'initialisation, de mettre en évidence les éventuels problèmes de cycle dans les liens entre instances et d'y remédier par des modifications de la spécification.

3.2.4.Un environnement "dédié" : Hypertext et dialogue H/M

L'environnement de spécification/maquettage OOA/KEE, à vocation "fonction nouvelle", a été utilisé pour aider à formaliser la fonction ELS (Electronic Library System) : ce travail a abouti à un outil de spécification/maquettage pour les applications ELS.

Une application ELS consiste à automatiser la consultation par le pilote des documents à bord de l'avion. Une telle application est destinée à être spécifiée en équipe intégrée avec la compagnie aérienne et les pilotes, d'où la nécessité de disposer d'un outil où spécification et maquettage sont indissociables.

L'outil permet de définir la structure logique de toute la base de données documentaire embarquée et la logique d'accès par le pilote à cette base de données. Le concept clé est celui d'"hypertext" : les documents sont reliés les uns aux autres par des liens typés. Une application ELS consiste en fait à naviguer dans la base de données documentaire en suivant les chemins d'accès définis par ces liens. Pour cela, le pilote dispose d'une interface utilisateur lui permettant grâce à un moyen de désignation (de type souris) de sélectionner des portions de documents et d'accéder à ceux auxquels ils sont reliés.

3.3.Atelier intégré de spécification / maquettage

L'atelier de spécification/maquettage regroupe différents environnements de spécification/maquettage et met à la disposition des utilisateurs des moyens répondant aux trois besoins d'intégration qui sont : intégration de techniques de maquettage entre elles, intégration de méthodes de spécification entre elles afin d'en définir de nouvelles et intégration d'une méthode avec une technique de maquettage.

Les travaux d'implémentation effectués jusqu'à présent dans le cadre de l'atelier de "spécification/maquettage" et présentés dans ce paragraphe ont conduit à des réalisations logicielles qui doivent être considérées comme des maquettes destinées à valider les besoins. Elles ont été réalisés directement sous UNIX alors que les moyens d'intégration qui seront effectivement retenus par la suite s'appuieront sur

des normes et produits du marché ; parmi ceux-ci, la norme PCTE et les outils supports (EMERAUDE), des environnements de type EAST ou ENTREPRISE sont à l'étude.

3.3.1.Structure d'accueil

3.3.1.1.Couche de communication

La croissance des besoins en communication qui s'explique par l'hétérogénéité des concepts liés aux méthodes de spécification et aux techniques de maquettage ainsi que par la distribution des outils correspondants sur un réseau de stations de travail, a nécessité la définition et la réalisation d'une couche de communication [14].

Grâce à cette couche, les moyens informatiques, logiciels et matériels, sur lesquels tournent les environnements de spécification/maquettage, deviennent transparents et le nombre et la localisation des stations de travail sur le réseau sont masqués au spécifieur.

Ceci est rendu possible par la faculté de faire communiquer des langages de programmation entre eux : ainsi l'interface entre ADA, KEE/CommonLisp, FORTRAN, C et LeLisp est réalisée par des primitives du type SEND(destinataire, message) et RECEIVE(émetteur, message). L'implémentation de ces primitives fait appel aux diverses fonctionnalités de la couche de communication d'UNIX, en particulier RPC et les SOCKETS.

L'intégration des techniques de maquettage consiste, à partir de la couche "communication entre langages", à définir des primitives de plus haut niveau : par exemple, la primitive ADA "ADD_FACT(base_de_connaissance,fait)" met à jour la base de connaissance d'un système expert tournant dans l'environnement KEE.

3.3.1.2.Repository commun

La structure d'accueil dispose d'un "repository" qui permet d'avoir une base de stockage des données commune à tous les environnements de spécification/maquettage. Dans cette base de données sont stockées des informations de tout niveau : on y trouve aussi bien les différentes spécifications et les maquettes associées que des éléments d'une granularité inférieure comme ceux figurant sur une spécification : par exemple, pour SART, on trouvera les flux de données, les fonctions, les PSPEC..., pour l'OOA, les classes, les relations, les automates...

Ce "repository" permet de faciliter l'intégration d'une méthode avec une technique de maquettage en disposant d'une représentation interne indépendante de la notation graphique ou textuelle de la méthode et à partir de laquelle le générateur de code est défini.

3.3.2.Intégration des méthodes

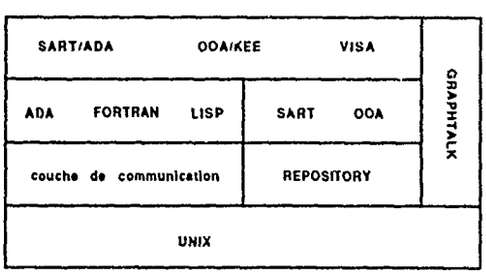
L'intégration de différentes méthodes se base sur l'existence, pour chaque méthode, d'une description formelle de sa syntaxe et de sa sémantique.

C'est à partir de la description formelle d'une méthode que sont définies :
- la représentation interne dans le repository commun des spécifications établies en utilisant la méthode en question,
- les vérifications, syntaxiques et sémantiques, liées à cette méthode.

La description formelle d'une méthode définit donc un outil support à cette nouvelle méthode.

L'intégration de deux méthodes consiste à enrichir les descriptions formelles de chacune des deux méthodes : on aura ainsi défini une nouvelle méthode intégrée ainsi que l'outil support associé et, grâce à la couche de communication, l'intégration des techniques de maquettage liées aux deux méthodes initiales sera assurée.

L'outil choisi pour supporter le travail de description formelle des méthodes est un outil du marché, GRAPHTALK réalisé par XEROX. A partir d'une définition graphique et textuelle d'une méthode (graphique pour la syntaxe et textuelle pour la sémantique), il permet la mise à jour du "repository" commun où les descriptions formelles des différentes méthodes sont elles-mêmes stockées. Les vérifications liées à la méthode résultant de l'intégration des méthodes exploitent ces descriptions formelles enrichies.



- Les différentes couches de l'atelier -

4. Conclusion

Ces travaux de définition et de mise en place d'outils de spécification et de maquettage ont, pour la plupart, été financés par des fonds propres SEXTANT. Ils ont largement bénéficié du soutien de plusieurs projets opérationnels, civils et militaires, dans le domaine de la conduite du vol : on peut citer le SOP (Système d'Optimisation des Performances) pour avions d'armes, la fonction ELMS (Emergency Landing Managment System), le projet PROFIL mené en collaboration avec l'Aérospatiale, l'ELS (Electronic Library System) et plusieurs applications des systèmes experts aux logiciels de gestion du vol.

Ceci a permis la définition de solutions parfaitement adaptées aux besoins et les outils associés sont mis en place selon un calendrier qui autorise leur utilisation dans le cadre d'importants projets actuels tel que le CET (Calculateur d'Elaboration de Trajectoires) du RAFALE.

De façon similaire à la phase de spécification/maquettage, une étude est menée concernant la phase de prototypage. Cette phase vise à valider une fonction dans son dialogue avec les systèmes existants à bord de l'avion : le principal problème posé par ce type de validation est de faire dialoguer une maquette fonctionnelle (dont le développement a fait abstraction des problèmes de performance liés au matériel) avec des équipements, réels ou simulés, nécessitant des temps de réponse proches de ceux attendus en vol.

Les deux principaux axes de cette étude des moyens de prototypage sont le passage automatique d'une spécification fonctionnelle ou de la maquette associée à un code exécutable en temps-réel et l'évolution des moyens matériels et logiciels pour recevoir le code et fournir l'environnement nécessaire à son exécution.

La compétence en spécification/maquettage, acquise lors des études et travaux liés à la mise en place de ces environnements, et la veille technologique assurée en permanence sur les techniques logicielles avancées permettent à la Division Conduite du Vol d'être prête à mettre sur pied très rapidement, l'environnement dédié à la spécification et à la validation par maquettage et/ou prototypage de toute nouvelle fonction que la division peut être amenée à développer.

5. Bibliographie

[1] Software Through Pictures, reference manual, IDE 88-89

[2] T. Demarco, "Structured Analysis and System Specification", Yourdon Press 78

[3] Ward & Mellor, "Structured Development for Real-Time Systems", Yourdon Press 85

[4] Pirbaï & Hatley, "Strategies for Real-Time System Specification", 88

[5] P. Coad, E. Yourdon, "Object-Oriented Analysis", Prentice-Hall 90

[6] KEE, Technical Manuals, Intellicorp 88-89

[7] HOOD, manuel de référence, version 3.0, ESA sept. 89

[8] D. Harel, "Statecharts : A visual approach to complex systems", Science of Computer Pogramming, Vol 8-3, 1987

[9] A. Gabriellan, R. Iyer, "Specifying Real-Time System with Extended Hierarchical Multi-State Machines", Thomson-CSF, INC. - PRO, 90

[10] Schlaer & Mellor, "Object-Oriented Systems Analysis", Prentice-Hall 90

[11] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall 88

[12] C. Delobel, M. Adiba, "Bases de données et systèmes relationnels", Dunod 82

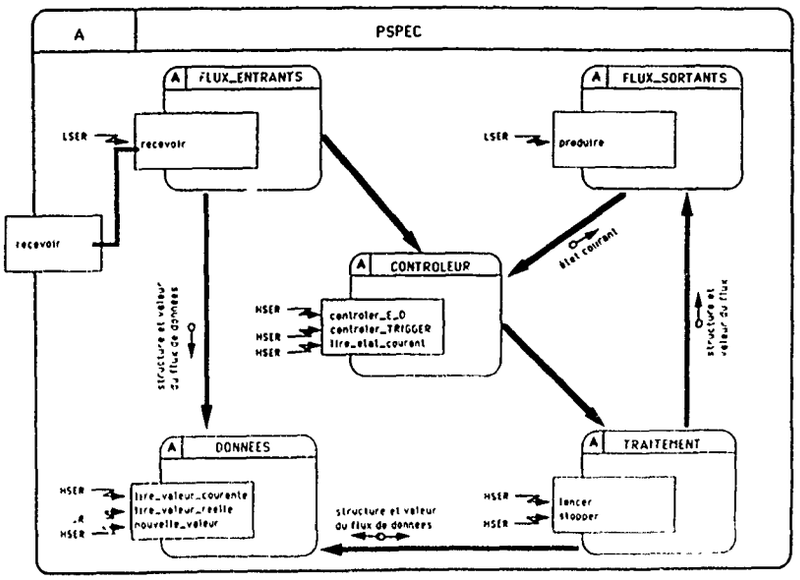
[13] D. Caignault, S. Gabison, Jl. Lebrun, "Atelier de développement de logiciels de pilotage-guidage", 52ème symposium AGARD, Thessalonik 91

[14] X. Chicot, "Environnement de communication pour le maquettage de nouvelles fonctions avioniques", Mémoire CNAM 91

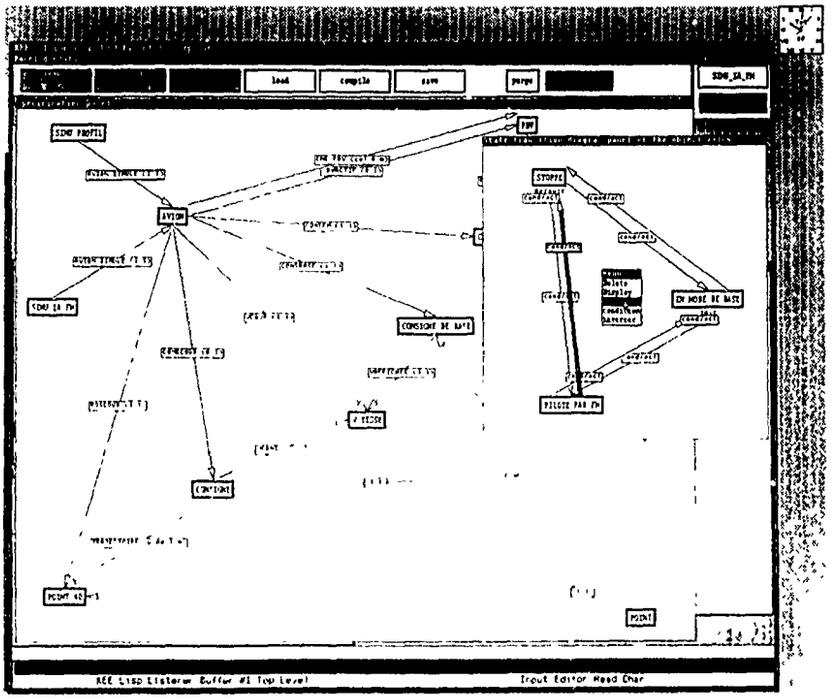
generic.ada

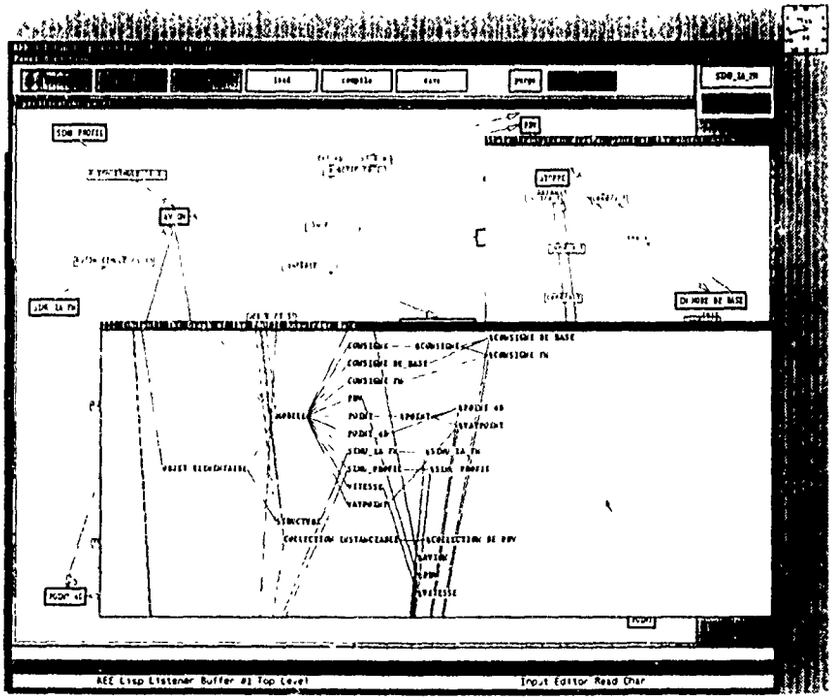
```
-------------------------------------------------------------------------------------------------------
--
--    connexion  et data_store
--
-------------------------------------------------------------------------------------------------------
with p_modele_sart;use p_modele_sart;
generic
    liste_flux_sortant : liste_de_flux,

package p_connexion is

task  tache_connexion is
    entry recevoir_flux ( un_flux : in flux);
end tache_connexion;

end p_connexion;

-------------------------------------------------------------------------------------------------------
--
--    pspec
--
-------------------------------------------------------------------------------------------------------

with p_modele_sart;use p_modele_sart;
generic
    with procedure traitement_pspec;
    liste_flux_sortant : liste_de_flux;

package p_pspec is

task gerer_entrees is
    entry recevoir_flux(un_flux : flux);
end gerer_entrees;

task  traitement;

task  controleur_local is
    entry controler_E_D;
    entry controler_TRIGGER;
    entry go;
    entry fin;
end controleur_local;

task  gerer_donnees is
    entry recevoir_donnees( un_flux : flux_de_donnees);
    entry valeur_courante ( val : out flux_de_donnees);
    entry valeur_reelle ( val : out flux_de_donnees);
end gerer_donnees;

task gerer_sorties is
    entry produire(un_flux : in flux);
    entry nouvel_etat(etat : in etat_fonction);
end gerer_sorties;

end p_pspec;

-------------------------------------------------------------------------------------------------------
--
--   cspec
--
-------------------------------------------------------------------------------------------------------

with p_modele_sart;use p_modele_sart;
generic
    l_automate : automate;
package p_cspec is

task  tache_cspec is
    entry recevoir_flux ( un_controle : in flux);
end tache_cspec;

end p_cspec;

-------------------------------------------------------------------------------------------------------
--
--   fonction
--
-------------------------------------------------------------------------------------------------------

with p_modele_sart;use p_modele_sart;
generic

flux_sortant_fonction : liste_de_flux;
flux_entrant_fils : liste_de_flux;
niveau_fonction : positive;

package p_fonction is

task tache_fonction is
    entry recevoir_flux ( le_flux : in flux);
end tache_fonction;

end p_fonction;
```

ANNEXE 1


A
PSPEC
A FLUX_ENTRANTS
LSER
recevoir
recevoir
A FLUX_SORTANTS
LSER
produire
état courant
A CONTROLEUR
HSER
HSER
HSER
controler_E_D
controler_TRIGGER
lire_etat_courant
structure et valeur du flux de données
structure et valeur du flux
A DONNEES
HSER
R
HSER
lire_valeur_courante
lire_valeur_reelle
nouvelle_valeur
structure et valeur du flux de donnees
A TRAITEMENT
HSER
HSER
lancer
stopper

ANNEXE 2


load
compile
save
purge
SDM PROFIL
AVION
STOPPE
CONSIGNE DE BASE
PILOTE PAR FN
CONSIGNE
POINT
REE Lisp Listener Buffer #1 Top Level
Input Editor Read Char


load
compile
save
purge
SDM PROFIL
AVION
STOPPE
CONSIGNE DE BASE
CONSIGNE FN
POINT
REE Lisp Listener Buffer #1 Top Level
Input Editor Read Char

ANNEXE 3

AGLAE - Atelier de Génie Logiciel de l'aerospatiale Engins

par
Jocelyne HAMON

aerospatiale
2, rue Béranger 92320 CHATILLON (FRANCE)

Résumé : La conception des systèmes temps-réel est un problème fortement combinatoire mais très contraint. Les experts procèdent par des transformations de la spécification technique de besoin en des systèmes équivalents, satisfaisant les contraintes du temps-réel (fréquence, retard, priorité, communications). **AGLAE** est un atelier logiciel comprenant une base de données orientée objets des composants et algorithmes utilisés ; une base de connaissances reproduisant le raisonnement de nos meilleurs experts ; un système de simulation permettant la validation des architectures matérielles et logicielles proposées. **AGLAE** est un programme de satisfaction de contraintes guidé par des heuristiques (relations de préférence des experts). La spécification forme la racine d'un arbre ET/OU où chaque nœud (fonctionnellement équivalent) correspond à l'application d'une transformation (ensemble de règles). Les feuilles de l'arbre final sont soumises à une simulation temps-réel. La recherche est dirigée par les dépendances qui sont générées comme dans un ATMS : les causes d'échec sont analysées et les conditions minimales sont retenues pour éviter la répétition de ces causes lors des retours.

I. INTRODUCTION :

Les systèmes de conception et de validation contiennent généralement des outils (qui ne sont pas nécessairement séparés) de synthèse, d'analyse et de test (simulation).

Un outil de synthèse (de système temps-réel) aide le concepteur dans la production d'une architecture matérielle, et d'un ensemble de programmes à partir d'une description de la fonction que doit remplir ce système et des contraintes liées aux données en entrée et en sortie.

Un outil d'analyse permet au concepteur de vérifier qu'un système remplit la fonction avec les performances désirées (il contient un vérificateur de règles de construction et de satisfaction de contraintes).

Un outil de simulation est indispensable pour s'assurer du respect des contraintes temps-réel, car l'arrivée des données peut avoir un caractère aléatoire (distribution dans le temps) que l'analyse ne peut pas toujours prendre en compte : il se peut que la simulation fasse apparaître des situations dans lesquelles une ou plusieurs des contraintes temps-réel sont violées. La spécification technique de besoin précise si ces violations sont acceptables de manière transitoire ou totalement prohibées. Dans ce cas, un autre système doit être conçu, puis simulé et ainsi de suite ...

L'atelier **AGLAE** ™[1] (Atelier de Génie Logiciel de **aerospatiale** Engins) décrit dans cet article, est un système expert qui reproduit le raisonnement d'un groupe de concepteurs de systèmes temps-réel. Il a pour but essentiel d'être un système de conception et de validation pour calculateurs embarqués ou tout autre système temps-réel.

1. AGLAE est une marque déposée par la société **aerospatiale**.

II. LES ATELIERS DE GENIE LOGICIEL :

Les progrès enregistrés ces quinze dernières années sur la production de composants matériels ont considérablement modifié le parc des systèmes informatiques. Parallèlement, si l'on examine l'évolution du développement du logiciel[2], on constate que l'effort s'est d'abord porté sur l'activité de codage, grâce aux langages de haut niveau et aux techniques de programmation structurée. Par contre, la qualité des documents de spécification et de conception reflète le plus souvent la valeur de l'analyste plutôt que la rigueur d'une méthodologie adaptée, ce qui conduit les chercheurs à aborder le développement du logiciel sous un nouvel angle. C'est ainsi qu'ont été identifiées puis mesurées les différentes activités de production du logiciel. L'existence même d'un cycle de vie standard a été reconnue et normalisée.

2. Le logiciel est l'ensemble des programmes, procédés et règles, et éventuellement de la documentation, relatifs au fonctionnement d'un ensemble de traitement de l'information (arrêté du 22.12.1981).

La figure 1 planche 1 montre le cycle de vie du logiciel proposé par l'Association Française pour le Contrôle Industriel de Qualité (AFCIQ).

Cette approche Génie Logiciel de la fin des années 70 s'est traduite par des tentatives d'intégration (au niveau de l'organisation du projet comme au niveau fonctionnel) de procédures, méthodes, langages et outils qui favorisent la production et la maintenance de composants logiciels de qualité.

Si l'on se réfère à l'arrêté ministériel du 30.12.1983, on appelle Génie Logiciel : "l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi."
Arrêté très important puisqu'il s'agissait alors de reconnaître l'existence du Génie Logiciel[1], connu seulement de quelques initiés. Vingt ans après, le Génie Logiciel fait l'objet de nombreux programmes nationaux et internationaux.

Selon P. JAULENT[2], le Génie Logiciel est un ensemble de "procédures, méthodes, langages, ateliers, imposés ou préconisés par les normes adaptées à l'environnement d'utilisation, afin de favoriser la production et la maintenance de composants logiciels de qualité."

En ce qui concerne les procédures, la mise en place d'une organisation de production de systèmes, qui permet au sein d'un cadre industriel, de maîtriser la qualité des produits, les coûts et les délais de réalisation repose sur le principe de découpage du processus de développement de systèmes en plusieurs phases. Ce découpage normalisé (DoD 2167, GAM T17, ...), appelé cycle de vie d'un système, est constitué de six phases.

Phase 1 : Orientation - Faisabilité des besoins
Cette étape décrit les besoins formulés par le futur client, et non comment les obtenir en tenant compte par exemple, des contraintes de coût, de délai, de temps d'exécution, de précision des calculs.
Le document créé lors de cette phase est appelé Cahier des Charges Fonctionnel (CdCF-1 norme AFNOR NFx50-151).

Phase 2 : Conception du système et validation des besoins
La conception d'un système suppose :
- une analyse statique, faite à partir du Cahier des Charges Fonctionnel, formalisée par une Spécification Technique de Besoins (STB),
- une analyse dynamique, faite à partir de la Spécification Technique de Besoins, formalisée par un prototypage.

L'analyse statique est composée de deux étapes : la conception préliminaire et la conception détaillée d'un système.

La conception préliminaire permet :
- d'énumérer et de décrire les travaux à effectuer,
- d'analyser les exigences de qualité,
- de définir la politique de maintenance,
- d'identifier les techniques de tolérance aux fautes à mettre en œuvre et étudier leurs conséquences,
- d'étudier les conditions de recette du système,
- de procéder à l'analyse fonctionnelle des besoins à partir du Cahier des Charges Fonctionnel, ceci afin de :
  * préciser les fonctions techniques et d'identifier les points critiques de chaque solution envisagée,
  * d'établir l'architecture du système (et des sous-systèmes).

D'un point de vue organisationnel, cette étape permet de définir la stratégie industrielle qu'il faudra appliquer sur le programme.

Les tâches entreprises au cours de l'étape de conception détaillée d'un système complètent, affinent et valident les aspects entrevus à l'étape précédente Celles-ci débouchent sur une nouvelle édition des Spécifications Techniques de Besoins

L'analyse dynamique des besoins maintenant exprimés, se traduit par la réalisation d'un prototype. Celui-ci doit nous offrir la possibilité d'expérimenter, rapidement et à moindres frais, le bien-fondé de certaines idées, qu'il s'agisse d'idées "fonctionnelles", d'idées architecturales, ou simplement d'idées concernant l'utilisation du système que l'on doit développer.

Il faut savoir qu'il existe deux types de prototypages : le prototypage rapide et le prototypage évolutif.

Le prototypage rapide consiste à réaliser tout ou partie du futur système avec des méthodes et des outils disponibles, et a pour but de vérifier la cohérence des diverses contraintes et de les préciser. Malheureusement, le coût et le temps de développement d'un tel prototype approchent souvent ceux d'une implémentation réelle.

Le prototypage évolutif est une partie intégrante du développement du système, quoique n'étant pas intégré au modèle classique de développement du logiciel, il se substitue aux étapes de conception et permet de vérifier les cohérences et les choix à ces niveaux.

Dans les deux cas, l'utilisateur du système peut évaluer le comportement du prototype, et le comparer à celui qu'il attendait. Si le prototype ne démontre pas les caractéristiques attendues, il peut éventuellement, après avoir identifié le problème, modifier sa spécification. Tout ceci lui permet également de ne pas prendre le risque d'attendre la fin de la réalisation du système pour s'apercevoir,

---

1 1968 : Conférence OTAN sur le Génie Logiciel , c'est l'année du constat de la crise du logiciel et l'utilisation pour la première fois de l'expression "Software Engineering".
2. Patrick JAULENT, Génie Logiciel les méthodes, Editions COLIN, 1990

à ce moment là, que les idées en question sont peut-être sujettes à caution ou tout simplement inadéquates.

Phase 3 : Développement logiciel du système
Cette phase est critique, puisqu'il s'agit de construire (ce qui n'est pas réalisable), suivant un cycle de vie normalisé, à partir des documents rédigés précédemment, l'ensemble des composants matériels et logiciels qui constitueront le système.

Sept sous-phases la composent :
- la spécification fonctionnelle du logiciel, donnant lieu à l'analyse des besoins exprimés par le client afin de définir le futur logiciel.

- la conception préliminaire du logiciel dont l'objectif est d'apporter une solution aux besoins exprimés en identifiant l'architecture du logiciel.

- la conception détaillée du logiciel où chaque composant identifié à l'étape de conception préliminaire fait l'objet d'une conception détaillée qui décrit les données manipulées par les composants et les algorithmes agissant sur ces données. Pour chaque composant, les tests unitaires lui afférant sont définis, afin de s'assurer que le composant réalisé répond à la description qui en a été faite.
Pour chaque composant, un document de conception détaillée et un document de tests unitaires sont élaborés.

- le codage du logiciel, où chaque composant logiciel, données ou algorithmes, est codé dans le langage de programmation choisi. Le code doit être compilé ou assemblé, puis "déverminé" soit par relectures, lectures croisées ou tout autre moyen de vérification.

- les tests unitaires de logiciel. Ici, pour chaque composant logiciel, les jeux d'essais définis dans la phase de conception détaillée sont exécutés. Les résultats sont enregistrés de même que tout écart par rapport aux résultats attendus.

- l'intégration et tests d'intégration du logiciel dont l'objectif est d'obtenir un ensemble intégré de composants logiciels de façon à constituer un produit final.

- la validation du logiciel, ou certification, dont le but est de démontrer que le logiciel développé répond exactement aux besoins exprimés dans la spécification.

Phase 4 : Intégration matériel-logiciel
Chaque entité ayant été construite et testée séparément, il est désormais possible de produire un système, en intégrant matériel et logiciel, de le tester et de le certifier conforme par rapport aux documents établis lors de la conception détaillée, puis de le fabriquer en vue d'une utilisation soutenue.

Phase 5 : Recette système - Validation
L'objectif de cette phase est de démontrer au client que le système développé répond effectivement aux besoins exprimés par le Cahier des Charges Fonctionnel et par les Spécifications Techniques de Besoins. Les tests de recette s'attachent à contrôler les caractéristiques telles que :
- les fonctionnalités du système,
- l'interface homme-machine (présentation des écrans, dialogues, résistance aux erreurs utilisateurs, ...),
- l'intégrité des données (protection),
- les temps de réponse,
- les reprises,
- les modes dégradés, ...

Phase 6 : Maintenance du système
Il s'agit de définir une politique de maintenance fiable pour le système sous la forme d'un contrat, en tenant compte des différentes catégories de maintenance (corrective, évolutive, adaptative, ...).

En ce qui concerne les méthodes, produire et maintenir un logiciel de qualité, en maîtrisant les coûts et les délais de développement, suppose l'utilisation d'une ou plusieurs méthodes pour les différentes phases du cycle de vie d'un système. Cependant, l'utilisation d'une méthode nécessite de bien maîtriser ses domaines d'application, ses possibilités, mais également ses limites, ses difficultés de mise en œuvre, ...
Il est absolument nécessaire, si l'on veut qu'une méthode apporte les gains de productivité escomptés, de la choisir en fonction de critères tels que :
- les finalités et stratégies de l'entreprise,
- les acteurs concernés,
- le domaine d'application (gestion, scientifique, contrôle de processus, ...),
- les étapes du cycle de vie du système couvertes par la méthode,
- le niveau d'outillage de la méthode (papier, intégré, industrialisé, ...).

Les principales méthodes aujourd'hui sont :
- SADT : Structured Analysis Design Technique (D.T. ROSS)
- SD : Structured Design (Rapport IBM : STEVENS, MYERS, CONSTANTINE)
- E-R : Entity-Relationship model (P. CHEN)
- SA : Structured Analysis (E.YOURDON, T. DEMARCO)
- JSD : Jackson System Development (M. JACKSON)
- SA-RT2 : Structured Analysis Real Time (I. PIRBHAI, D. HATLEY)
- HOOD : Hierarchical Object Oriented Design (BOOCH, MATRA, CRI, CISI)
- ...

Pour les langages, plusieurs tendances se dégagent actuellement dans le monde informatique :
- la programmation algorithmique construite dans le domaine du procédural (FORTRAN, PASCAL, C) ou du modulaire (ADA),
- la programmation purement fonctionnelle et déclarative (LISP),
- la programmation par objets (C++, SMALTALK),

- la programmation logique (PROLOG).

Chaque entreprise, en fonction de son expérience ou de ses objectifs choisit l'un ou l'autre langage. Mais il faut noter, à l'heure actuelle, l'apparition d'outils tels que des générateurs de code permettant d'automatiser au maximum cette étape de programmation.

Les ateliers de Génie Logiciel, quant à eux, supportent la mise en place d'une organisation industrielle de production et de maintenance de logiciels en regroupant harmonieusement :

- les méthodes reconnues telles que SADT, SA-RT, JSD
- les procédures de développement normalisées comme DoD 2167, GAM T17
- les générateurs de code PASCAL, C, ADA
- les outils tels que :
  * des gestionnaires de projets,
  * des interfaces avec un gestionnaire de configurations,
  * des modèles d'estimation des coûts,
  * des outils de qualimétrie et de maintenance du logiciel,
  * des interfaces PAO.

Ils poursuivent de nombreux objectifs dont les principaux sont :

- d'augmenter la productivité d'une équipe de développement,
- d'améliorer la qualité des produits logiciels, c'est-à-dire leur fiabilité, leur évolutivité, leur maintenabilité,
- d'aider l'équipe à appliquer les différentes normes et procédures, incontournables étapes dans le processus de développement,
- de soulager l'équipe de tâches fastidieuses et répétitives telles que les vérifications de cohérence lors des phases de spécification et de conception du logiciel, ...

III. LA CONCEPTION DES SYSTEMES TEMPS-REEL :

La conception d'un système (conceptions préliminaire et détaillée) est l'étape la plus délicate du cycle de vie d'un système, puisqu'elle suppose de la part de l'architecte, des compétences sur les deux principales composantes en interaction : le matériel et le logiciel. En effet, c'est effectivement lors de cette étape que s'effectuent les choix de ce qui sera fait en matériel et en logiciel, mais également, avec quel type de matériel et de logiciel sera construit le futur système.

Malgré les nombreux moyens qui permettent aujourd'hui de concevoir l'architecture d'un système, nous avons délibérément choisi de développer notre propre méthode, et ceci pour les raisons suivantes :

- les méthodes proposées sur le marché ne couvrent pas suffisamment toutes les contraintes exprimées dans le domaine du temps-réel,
- il nous faut un produit approchant au maximum le raisonnement de nos experts dans ce domaine et tenant compte de leurs expériences et de leur savoir-faire.

La Direction des Etudes, au sein de la Division Engins Tactiques de **aerospatiale**, dans son Etablissement de Chatillon, a réalisé un prototype d'atelier de Génie Logiciel répondant à ses besoins. Cet atelier, nommé **AGLAE** (Atelier de Génie Logiciel de **aerospatiale** Engins) a pour objectif d'automatiser les étapes de conceptions préliminaire et détaillée, en proposant au concepteur un ensemble de solutions, architectures matérielles et architectures logicielles, adaptées à son problème : élaboration d'un calculateur embarqué ou de tout autre système soumis à des contraintes temps-réel.

Si nous comparons notre démarche avec celle définie dans le chapitre précédent, nous pouvons indiquer que chacune des solutions proposées par l'atelier est en fait un prototype évolutif de la partie fonctionnelle du système (sans les interfaces et l'environnement). L'utilisateur peut donc ainsi valider sa spécification, à partir du comportement du système simulé par **AGLAE**. Si aucune solution n'a pu être produite, alors, l'utilisateur peut modifier sa spécification et continuer sa recherche.

A partir de ces remarques, une vue plus détaillée du cycle de vie s'impose (§ figure 2 planche 1).

A l'origine de cette phase de conception, le document relatif à la Spécification Fonctionnelle Technique doit être rédigé. Celui-ci détient de nombreuses informations qu'il nous importe de connaître avant l'utilisation de l'atelier **AGLAE**.

1. Spécification Fonctionnelle Technique :

Tout système informatique comprend une architecture matérielle (processeurs, bus, mémoires, coupleurs, ...) et un ensemble de logiciels qui, exécutés sur cette architecture, réalisent une fonction (au sens mathématique du terme). Cette fonction est généralement décrite, dans la Spécification Technique de Besoins du système, comme une composition d'autres fonctions (qui sont elles-mêmes des compositions de fonctions et ainsi de suite). Elle est encore appelée Spécification Fonctionnelle Technique du système

Dans un système temps-réel, la fonction attendue est généralement de contrôler et de commander un processus. Pour cela, l'exécution des programmes est toujours commandée par les données ; certaines données en entrée doivent être prises en compte dans un délai très court (fenêtres d'entrée) ; certaines données en sortie doivent être produites à un instant donné (fenêtres de sortie). Les contraintes, liées à la nature des données en entrée et en sortie, peuvent ainsi s'exprimer en termes de fréquence, de retard maximum à la prise en compte,d'interruptions, de date de production, de dates de consommation minimales et maximales, ... Par ailleurs, d'autres contraintes peuvent être imposées, comme la qualité requise pour le système (choix de composants, d'architectures, ...) et la précision des calculs.

La conception de l'architecture matérielle est similaire à la construction d'un circuit électrique ou électronique simple, et est totalement pris en charge par **AGLAE**. Au contraire, les logiciels, eux, ne sont pas conçus par **AGLAE**, le code des algorithmes de base (fonctions élémentaires) figure dans la Spécification Fonctionnelle Technique.
L'organisation du logiciel sur le matériel peut ainsi être vu comme un ordonnancement, mais à la différence de celui d'un atelier, les tâches sont interruptibles et du fait des asservissements (boucles où, pour deux programmes, les données produites par l'un à un instant t sont consommées par l'autre à un instant t', avec t supérieur à t'), il s'avère nécessaire de gérer des communications et des rendez-vous à dates fixes. De plus, cette organisation est telle qu'elle doit vérifier, à la fois la fonction demandée, mais également les contraintes exprimées (le plus souvent non relaxables).

Le problème a été étudié sous tous ses aspects (objets, moniteurs, parallélisme, ...). Les résultats sont utilisés dans les connaissances d'**AGLAE**, soit pour décrire le système, soit sous la forme de contraintes à respecter, soit sous la forme de règles de transformation de systèmes.

2. Exemple :

Soit la Spécification Fonctionnelle Technique, figure 3 planche 2.

La fonction PHASE-VOL est définie comme une composition des fonctions PILOTAGE et RECALAGE.
La fonction PILOTAGE est elle-même décomposée en deux sous-fonctions :
- PILOTAGE-LENT,
- PILOTAGE-RAPIDE.

Ces deux fonctions se distinguent par leur fréquence dont l'une est plus rapide que l'autre.
Les fonctions produisent et consomment différentes données :
- PILOTAGE-LENT consomme la donnée POS-MACH et produit la donnée GAIN,
- PILOTAGE-RAPIDE consomme les données :
  * GAIN produite par PILOTAGE-LENT,
  * ACC-COMM produite par la fonction RECALAGE,
  * SOUS-CYCLE-0 et SOUS-CYCLE-1 produites par les ressources externes (bus d'entrée-sortie).

Cette fonction produit la donnée nommée INCIDENCE.

La fonction de RECALAGE consomme les données :
* SOUS-CYCLE-1 issue comme précédemment d'une ressource externe,
* INCIDENCE produite par PILOTAGE-RAPIDE.
  Il faut noter qu'INCIDENCE est une donnée asservie, c'est-à-dire que la donnée utilisée par RECALAGE provient de la période précédente. Ceci se fait généralement quand la mise à jour des données n'est pas primordiale pour les nouveaux calculs, ou quand cette donnée n'a pas le temps d'être prise en compte dans la période même où elle est produite.

Trois données sont produites par RECALAGE :
* SORTIE-1-COMM envoyée sur une ressource externe,
* ACC-COMM et POS-MACH absorbées par PILOTAGE.

Toutes ces données ne sont produites ou consommées qu'à certains moments du cycle (période). Ces instants sont appelés "fenêtres" et sont matérialisés par les ressources externes ENTREE-0, ENTREE-1 et SORTIE-0.

Le problème, pour une telle spécification, est de faire exécuter l'ensemble des fonctions de telle sorte que tous les rendez-vous soient tenus. Ce sont les contraintes du temps-réel. Une solution possible, dépendante du temps d'exécution de chaque fonction (matérialisé par un créneau grisé), est décrite dans les figures ci-après.

(§ figure 4 planche 2 : Découpe du logiciel durant une période courte)

Durant cette période courte, les deux fonctions RECALAGE et PILOTAGE-RAPIDE sont exécutées l'une après l'autre, la fonction PILOTAGE-RAPIDE nécessitant la fourniture d'une donnée de RECALAGE.
Sur ce graphique, les instants d'entrée et sortie de données externes sont matérialisées par les pics en lecture (r) et en écriture (w).

Durant une période longue, égale dans cet exemple à huit périodes courtes, la fonction PILOTAGE-LENT est réalisée.

(§ figure 5 planche 2 : Découpe du logiciel durant une période longue)

Il faut noter que le temps d'exécution de cette dernière fonction a permis de l'effectuer en deux fragments durant la première période courte. Il aurait été possible, si nécessaire, de la segmenter encore pour la réaliser sur plusieurs périodes courtes.

L'exemple décrit ici a permis de citer un petit échantillon de contraintes à prendre en compte lors de la réalisation d'un système temps-réel. Il nous faut maintenant aller plus loin et décrire non seulement la solution obtenue mais aussi le raisonnement appliqué par l'expert pour l'obtenir, ce qu'**AGLAE** reproduit automatiquement.

IV. L'ATELIER LOGICIEL **AGLAE** :

1. Système expert :

La réalisation d'un système expert nécessite un important investissement (temps et coût). Aussi, pourquoi avons-nous choisi cette solution ?
Les raisons sont les suivantes :
- réussir à conserver dans l'entreprise une connaissance

et un savoir-faire. Le problème est d'autant plus aigü que les personnes détenant la connaissance sont dans notre cas peu nombreuses. De plus, le système expert n'est pas un stockage passif de la connaissance mais il est constamment remis à jour et enrichi,
- permettre aux experts de se décharger des tâches répétitives et de se consacrer à d'autres travaux,
- diminuer les coûts de mise au point puis de maintenance d'une application, ...

Ces considérations sont très générales et applicables à tout système expert. Maintenant, nous pouvons ajouter que cette méthode nous permet dans le cas d'**AGLAE** de résoudre des problèmes fortement combinatoires (nombre important de solutions possibles) et également très contraints (dû aux contraintes temps-réel).

D'un point de vue structure, les systèmes experts se caractérisent par leur architecture. A la différence de la programmation classique qui oppose le programme aux données, trois composantes sont identifiées dans un système expert :
- la base de faits,
- la base de règles,
- le moteur d'inférence.

Une quatrième composante existe :
- l'interface homme/machine (§ chapitre V Résultats).

Avant de décrire de façon précise l'implémentation d'**AGLAE**, donnons quelques définitions :

- la base de faits contient l'ensemble des données propres au problème à résoudre. Cet ensemble évolue au cours de la résolution du problème en intégrant les résultats intermédiaires et la ou les solutions obtenues par le système. La base peut s'enrichir en cours d'exécution si le système fait appel à l'utilisateur pour qu'il introduise de nouvelles données.

- la base de règles, encore appelée base de connaissances, est constituée par l'ensemble des méthodes de résolution du problème déterminé. Pour communiquer ces méthodes au système, l'expert utilise une technique de représentation de la connaissance : le plus souvent, il s'agira de règles ou productions de la forme :

  Si conditions Alors actions

  La base de connaissances est ainsi constituée d'un ensemble de ces entités élémentaires : les règles. Celles-ci sont déclaratives, elles traduisent le savoir-faire et l'expérience de nos experts et spécialistes du domaine sans préjuger de leur utilisation.

- le moteur d'inférence est un mécanisme général de raisonnement chargé de résoudre le problème spécifié par la base de faits à l'aide des connaissances contenues dans la base de règles. Le moteur d'inférence est un logiciel accessible, ni à l'utilisateur final, ni à l'expert.

A partir de ces définitions, reprenons chaque module et regardons comment il a été réalisé dans **AGLAE**.

2. Base de faits :

Dans **AGLAE**, la base de faits est décomposée en deux sous-ensembles :
- tout d'abord, une base de faits propres au problème à résoudre, comme décrite précédemment, reprenant la description de la Spécification Fonctionnelle Technique (mission et contraintes du système),

- puis, la base de faits permanents, ou encore appelée base de données, comprenant toutes les caractéristiques techniques et commerciales des matériels utilisables (composants du marché, composants militarisés, ...).

2.1. Spécification Fonctionnelle Technique :

2.1.1. Pratique des experts :

Les experts distinguent deux types d'objets, les fonctions et les données :

- les fonctions, encore appelées Machines Abstraites (MA), représentent soit des éléments matériels, soit des éléments logiciels du futur système,

- les données forment l'ensemble des communications virtuelles ou réelles entre les fonctions. Une donnée est produite pour une unique fonction mais peut être consommée par plusieurs.

A chaque donnée sont associées des informations telles que les fonctions qu'elle relie, sa précision, ...
A chaque fonction sont associées plusieurs informations, telles que son entrée (données consommées), sa sortie (données produites), le traitement qu'elle réalise, sa fréquence, son temps d'exécution, sa précision, ...

Maintenant, les données et les fonctions énoncées ici, ne suffisent pas à décrire toutes les contraintes dues au temps-réel. Comme nous l'avons vu sur l'exemple précédent, il faut aussi décrire, à partir de la spécification, des coupleurs d'entrée-sortie qui, à chaque période, représentent des fenêtres ouvertes à un certain moment pendant une certaine durée.
Ces fenêtres imposent que les fonctions qui produisent ou consomment ces données soient exécutées (pour faire leurs entrées et sorties au moins) pendant la durée de ces fenêtres.
Ces fonctions seront représentées dans **AGLAE** comme des machines abstraites, mais avec des caractéristiques supplémentaires telles que les dates de début et de fin de production, les dates de début et de fin de consommation de données.

2.1.2. Modélisation dans **AGLAE** :

L'ensemble de la Spécification Fonctionnelle Technique est, comme nous venons de le voir, composée uniquement de fonctions et de données. Ces deux éléments sont considérés dans **AGLAE** comme des objets à part entière et matérialisés par des schémas compatibles

avec le formalisme du générateur utilisé.

Pour les fonctions, un ensemble de machines abstraites (MA) est créé. Chaque MA est une instance d'un objet de la base de faits et est décrite sous la forme suivante :

Schéma PHASE-VOL
instance machine-sequ (pour machine décomposable)
décomposition-en ENTREE-0 ENTREE-1
PILOTAGE RECALAGE SORTIE-0
est-un-composé-de (nom de la machine parent)
unité-temps ms
nbre-unité/s 1000
entrée (pour données en entrée)
sortie (pour données en sortie)
...

Une des machines la composant est RECALAGE :

Schéma RECALAGE
instance machine-base (pour machine abstraite)
est-un-composé-de PHASE-VOL
temps-exécution-maximum 10 (pour 10 ms)
entrée (SOUS-CYCLE-1 0) (SOUS-CYCLE-1 -1)
sortie (POS-MACH 0) (ACC-COMM 0)
(SORTIE-1-COMM 0)
(0 pour standard, -1 pour asservissement)
fréquence 60 (en Hertz)

Une des données produite par RECALAGE est POS-MACH :

Schéma POS-MACH
instance paquets-donnees
prod-paquet RECALAGE (fonction productrice)
cons-paquet PILOTAGE-LENT
(fonction consommatrice)

Tous ces objets sont réunis dans des fichiers modifiables par l'utilisateur, soit de façon textuelle (éditeur ASCII), soit sous forme graphique (interface spécifique).

2.2. Base de données Matériel :

2.2.1. Pratique des experts :

Le second sous-ensemble composant la base de faits est chargé de détenir l'ensemble des informations matérielles.
Il s'agit à la fois :
- des types d'architectures, mono et multi processeurs, simples et complexes avec mémoires couplées, DMA[3], 1 à N bus externes, ...
Celles-ci ont été recensées parmi les architectures les plus communément utilisées dans l'entreprise mais aussi parmi celles connues à ce jour.

- des différents composants matériels (bus, processeurs, mémoires, coupleurs, ...). Ces composants sont, ou bien spécifiques (développés en interne à **aerospatiale**, ou bien du commerce). Dans ce dernier cas, toutes les informations sont accessibles à partir des documentations des divers fournisseurs.

2.2.2. Modélisation dans **AGLAE** :

De la même façon que la base de faits relative à la spécification, les différentes architectures et composants sont entrés dans **AGLAE** sous la forme d'objets. Par exemple, l'architecture ARCHI-MONO-COUPLEE (architecture mono-processeur avec mémoires couplées) se présente sous la forme d'un schéma :

Schéma ARCHI-MONO-COUPLEE
is-a mono-processeur
processeur (nombre de processeurs)
bus-interne (nombre de bus)
mémoire-donnée (nombre de mémoires de données)
mémoire-programme (nombre de mémoires de code)
transfert-données-int
(pour temps de transfert des données en interne)
transfert-programme (temps de transfert du code)
majoration-accès-mémoire (temps accès mémoire estimé s'il n'est pas connu exactement)

Quant aux matériels, la description d'un processeur se fait de la façon suivante :

Schéma PROCESSEUR-X
is-a processeur (appartient à la classe des processeurs)
fréquence 1E6 (pour 1 Méga-Hertz)
+ 4 (l'opération addition dure 4 cycles de base)
... (temps respectifs pour les autres opérations)
coprocesseur (liste des coprocesseurs associés)
... (liste des autres composants associés)

Pour les autres composants, les caractéristiques propres sont inscrites sous la forme de propriétés relatives à l'objet décrit.

Globalement, tous les objets composant les deux sous-ensembles de la base de faits sont réunis dans une structure arborescente comprenant, à partir d'une racine, différentes classes :
- la classe RACINE-MATERIEL ou base de données Matériel constituée de l'ensemble des types d'architectures et des composants matériels,
- la classe RACINE-LOGICIEL décrivant la Spécification Fonctionnelle Technique composée des machines abstraites et des données,
- la classe RACINE-SOLUTION où sont créés, durant la résolution, les objets relatifs aux solutions proposées (matériel et logiciel).

(§ figure 6 planche 3)

3. DMA : Direct Access Memory ou Accès direct à la mémoire.

3. Base de connaissances :

Si le moteur d'inférence (§ paragraphe 4 Moteur d'inférence) représente "l'intelligence" d'**AGLAE**, les règles, elles, représentent son unique "connaissance". Extraites de l'expérience des experts et spécialistes du domaine, elles permettent au système de trouver, par itérations successives la solution recherchée.

Elles sont formalisées de la façon suivante :
Si conditions Alors actions

Aujourd'hui **AGLAE** contient onze classes de règles :

- architecture : choix d'une architecture matérielle en fonction des contraintes de la spécification,

- composants : complétion de l'architecture matérielle choisie grâce à la base de données des composants,

- durée : calcul du temps d'exécution de toutes les fonctions à partir de la durée des opérations élémentaires du processeur choisi,

- coût : respect du seuil imposé par l'utilisateur concernant le calcul du coût des composants choisis,

- chemins : recherche des chemins de données prioritaires,

- communications : création des machines abstraites de communications sur tous les chemins de données,

- séquencialisation : création d'une chaîne temporelle regroupant l'ensemble des machines abstraites,

- datation : calcul des temps relatifs à chaque fonction dans la chaîne temporelle,

- découpage : segmentation des fonctions lentes sur plusieurs périodes courtes,

- déplacement : en cas d'échec, on effectue une modification de l'emplacement d'une fonction dans la chaîne temporelle,

- déphasage : en cas d'échec, les fonctions ne pouvant être exécutées dans la période impartie devront accepter les données de la période précédente.

Par exemple, nous aurons :

Une règle de gestion temporelle des ressources :
Si deux machines abstraites ont des fréquences différentes,
Alors la machine abstraite de fréquence la plus rapide est prioritaire.

ou encore une règle de regroupement :
Si deux machines abstraites ont la même fréquence et que l'une produit des données exclusivement pour l'autre,
Alors créer une machine abstraite les regroupant.

4. Moteur d'inférence :

4.1. Principe :

Pour **AGLAE**, un moteur d'inférence spécifique a été construit.
Les systèmes fonctionnant en chaînage arrière (comme PROLOG) sont inadéquats pour le genre de problèmes traités par **AGLAE** (construction ou synthèse).
Les systèmes classiques en chaînage avant (comme OPS) sont peu efficaces et posent de nombreux problèmes liés à la non-monotonie (une règle peut détruire un fait sur lequel reposent d'autres faits).
Les problèmes de construction étant des problèmes dans lesquels les choix sont nombreux, il est plus aisé de travailler sur des contextes séparés (un par choix) et de maintenir, d'un contexte père à ses fils, la cohérence des informations au moyen d'un programme spécial : un TMS[4].
Or, la complexité d'une tâche de résolution de problèmes est fonction, à la fois du nombre de règles exécutées, et du nombre de contextes considérés dans la recherche.

L'une des premières raisons qui a motivé la construction d'un nouveau moteur pour **AGLAE** a été la nécessité de spécifier le contrôle du besoin exprimé par l'ordre de déclenchement des règles, et exprimant le raisonnement des experts. Un tel contrôle, réalisé par un programme procédural classique, diminue la flexibilité de l'outil et complique sa maintenance.
Dans **AGLAE**, le contrôle est traduit par la réunion de certaines règles en classes (sources de connaissances), et par l'utilisation de méta-règles pour le choix de ces sources de connaissances dans un contexte donné.
Les contextes forment ainsi un graphe (arbre) qui est exploré suivant une procédure BRANCH AND BOUND généralisée (GBB[5]). Celle-ci incorpore à la fois l'utilisation :
- d'un ordre de préférence sur les sources de connaissances et les règles,
- d'un "backtracking" intelligent (DDB[6]),
- d'un ATMS[7] pour maintenir la cohérence de chaque contexte.
En effet, un résolveur de problème contrôlé par DDB a tendance à être plus efficace pour des problèmes dans lesquels quelques solutions parmi toutes celles possibles sont désirées. L'efficacité est obtenue en organisant la recherche, de manière à ne trouver qu'une solution spécifique d'abord. La combinaison d'un ATMS et de DDB, à la fois, réduit le nombre de contextes examinés et de règles exécutées, et permet d'obtenir efficacement une solution spécifique en premier.

---

4. TMS : Truth Maintenance System
5. GBB : Generalized Branch and Bound
6. DDB : Dependency Directed Backtracking
7. ATMS . Assumption Truth Maintenance System

Quand une contradiction est rencontrée, le retour (backtracking) est effectué jusqu'au premier "générateur" contribuant à la contradiction (c'est-à-dire la génération de la première cause d'échec). Toutes les raisons pour éliminer des valeurs sont combinées pour former une contradiction. Mais le système ne se souvient pas de ces contradictions quand une autre branche est trouvée. Certaines d'entre elles peuvent donc être calculées plusieurs fois de suite. Dans **AGLAE**, chaque cause d'erreur (conjonction minimale) est retenue de manière permanente pendant la recherche.

4.2. Modélisation dans **AGLAE** :

En s'appuyant sur l'algorithme précédent, **AGLAE**, à partir de la spécification fonctionnelle, doit commencer par construire un système fonctionnellement équivalent. L'architecture matérielle est, soit celle qui est imposée, soit la plus simple possible. L'organisation logicielle correspond exactement à la spécification : pas d'interruptions, exécution des tâches entièrement séquentielle. Si, après simulation, le résultat est acceptable selon les contraintes temporelles, il est proposé.

Dans le cas contraire, cette architecture constitue alors la racine de l'arbre des solutions. Les transformations entre un nœud père et ses fils résident dans l'organisation temporelle des tâches et la suppression ou l'ajout éventuel de composants matériels.

La réorganisation temporelle se fait par découpage et regroupement des sous-fonctions de la spécification, le but final étant de :
- respecter les retards sur les données,
- diminuer le temps global de calcul.

Tout nœud est alors testé comme une solution éventuelle du système. En cas d'échec, on isole la cause minimale de l'échec et on supprime tous les nœuds de l'arbre vérifiant cette cause.
Si aucune des organisations essayées ne satisfait l'ensemble des contraintes, une autre architecture matérielle est recherchée, ou une modification des fréquences est imposée.

V. RESULTATS :

Les figures ci-après montrent pour l'exemple décrit au début du document, la solution proposée par **AGLAE**. On retrouve sur la planche 4, la Spécification Fonctionnelle Technique sous forme graphique.
Puis, sur cette même planche, les chronogrammes résultats sur une période courte et une période longue.

En plus des fonctions, dont l'exécution est matérialisée par des créneaux, les chronogrammes indiquent :
- le nom des fonctions de communications créées,
- l'instant où elles sont utilisées à des fins de mémorisation de données (en écriture ou en lecture),
- les fonctions de gestion de contextes permettant la sauvegarde et la restauration de l'environnement lors d'interruptions ou de reprises de fonctions lentes.

La dernière planche indique l'ensemble des composants choisis, leur coût, l'architecture matérielle adoptée ainsi que le résultat complet de la simulation du système avec, pour la tâche rapide (période courte) et la tâche lente (période longue) :
- le temps total d'exécution,
- le temps total de communications interne et externe,
- le temps total de gestion des contextes,
- le taux de charge du processeur choisi.

Il faut noter que cette dernière information est primordiale pour le concepteur, car elle le renseigne sur les limites du système en ce qui concerne les modifications de maintenance (possibilité d'augmentation de la taille des logiciels, de la vitesse des processeurs, du changement de composants, ...).

Tous ces résultats sont présentés sur l'écran **AGLAE**. Afin de rester le plus convivial possible, aucune commande n'est entrée sous forme textuelle. Toute demande de traitement se réalise grâce à la souris en "cliquant" sur une icone à droite de l'écran.

Six icones sont représentées :
- la première permet l'affichage d'un texte de présentation du logiciel,

- GEST pour Gestion
  * LOAD : chargement d'une spécification,
  * SAVE : sauvegarde sur disque d'une spécification,
  * CHECK : vérification syntaxique et sémantique d'une spécification,
  * KILL : effacement d'une spécification en mémoire,
  * RUN : résolution et recherche de solutions,
  * QUIT : sortie d'**AGLAE**.

- VIEW pour fonctions de Visualisation
  * OVERVIEW : spécification complète sur la page écran,
  * ZOOM : vue limitée d'une partie de la spécification avec déplacements grâce à des ascenseurs verticaux et horizontaux le long de la fenêtre de visualisation,
  * TRAME : Trame de fond ou écran d'attente.

- EDIT pour Edition de la base de données
  * EDIT : édition des objets stockés dans **AGLAE**,
  * TREE : visualisation des objets et des classes sous la forme d'arbres.

- GRAPH pour éditeur Graphique de la spécification
  * COMPOSE : crée des fonctions décomposables,
  * FUNCTION : crée une machine abstraite de base,
  * WINDOW : crée une fenêtre d'entrée-sortie,
  * DATA : crée une donnée,
  * ARCHITECTURE : crée l'objet descripteur de l'architecture choisie si elle existe,

* DELETE : permet d'effacer une boîte ou une donnée,
* MOVE : permet de déplacer une boîte ou une donnée sur l'écran.

- LASER pour impression sur un périphérique externe
  * ECRAN : image écran en format POSCRIPT,
  * SOLUTION : texte associé à la résolution avec propositions de solutions,
  * ECHEC : texte associé au refus de solutions avec le diagnostic prononcé.

VI. CONCLUSION :

1. Validation :

Plusieurs systèmes opérationnels sont utilisés pour la validation des connaissances contenues dans **AGLAE**, ceci dans le but de vérifier :
- qu'avec la seule description de la Spécification Fonctionnelle Technique, **AGLAE** est à même de proposer une voire plusieurs solutions,
- qu'avec un même choix de composants, les solutions proposées sont similaires à celles retenues par les concepteurs humains.

Il est important de préciser que ces solutions sont obtenues en quelques minutes (après l'entrée des données), alors que l'élaboration et la validation d'une autre solution prenait auparavent plusieurs jours aux ingénieurs.

2. Evolutions :

AGLAE est écrit en Common LISP et s'appuie sur le générateur de système expert KNOWLEDGE CRAFT™[8] V3.3. Les objets sont des schémas CRL™[9], tandis que les autres connaissances (règles) et le moteur d'inférence sont des fonctions de Common LISP.
Le matériel utilisé est un SUN 3/260™[10] sous environnement UNIX™ V4.0.3[11].

Malgré les quelques 110 objets répartis dans les bases de faits, et les 11 grands sous-ensembles de règles de la base de connaissances, **AGLAE** est encore loin de résoudre tous les problèmes temps-réel rencontrés dans notre Division.
Les évolutions prochaines du produit permettront par exemple :
- l'enrichissement de la base de données pour prendre en compte un nombre plus important de composants du marché et de nouvelles architectures,
- l'enrichissement de la base de connaissances pour offrir des solutions à base de multi-processeurs. Ceci implique l'analyse fine des différents parallélismes possibles (étude des grains),
- le couplage avec des ateliers de Génie Logiciel du commerce qui permettra de couvrir l'ensemble des phases du cycle de vie d'un système notamment :
  * la phase de codage du logiciel (ADA, C, ...),
  * la phase de test automatique du logiciel codé,
  * la phase de documentation sous des formats normalisés tels que AQAP 13, GAM T17, DoD 2167B, ...

Quant à la phase de spécification, une étude est en cours afin de faciliter le travail du concepteur humain. En effet, **AGLAE** devra lui-même afficher la Spécification Fonctionnelle Technique sous forme graphique, à partir d'un texte en langage naturel, ceci pour limiter au maximum les manipulations d'éditeurs toujours fastidieuses pour l'utilisateur final.

Enfin, un portage en langage C++ est envisagé avec un générateur d'interfaces suffisamment normalisé pour envisager une utilisation de l'atelier **AGLAE** sur des types de console et d'environnement quelconques.

8. KNOWLEDGE CRAFT est une marque déposée par CARNEGIE GROUP INC.
9. CRL : CARNEGIE Representation Language est une marque déposée par CARNEGIE GROUP INC.
10. SUN est une marque déposée par SUN MICROSYSTEMS.
11. UNIX est une marque déposée par AT&T BELL LABORATORIES.

BIBLIOGRAPHIE :

BRUYNOOGHE M., Solving combinatorial search problems by intelligent backtracking, Information Processing Letters 12, 1981, pages 36-39.

BRUYNOOGHE M. - PEREIRA L.M., Deduction revision by intelligent backtracking, dans CAMPBELL J.A. (Ed), Current issues in Prolog implementation, New York, Wiley, 1984, pages 194-215.

BENAY G. - VAZEILLES M. - VILLEMIN F.Y., Conception intelligemment assistée de systèmes temps-réels, **aerospatiale** & CNAM-CEDRIC, Mémo n°491, 1989.

de KLEER J., An assumption-based truth maintenance system, Artificial Intelligence, vol.28, n° 1, 1986, pages 127-162.

de KLEER J., Extending the ATMS, Artificial Intelligence, vol.28, 1986, pages 163-196.

de KLEER J.,Problem solving with the ATMS, vol.28, 1986, pages 197-224.

de KLEER J.- WILLIAMS B.C., Back to backtracking : controlling the ATMS, Engineering Automated reasoning, 1987

JAULENT Patrick, Génie logiciel les méthodes, Paris, Armand Colin Editeur, 1990, pages 14-34 et 258-278.

LUQI, Software evolution through rapid prototyping, Computer, Mai 1989.

8 - Planche 1

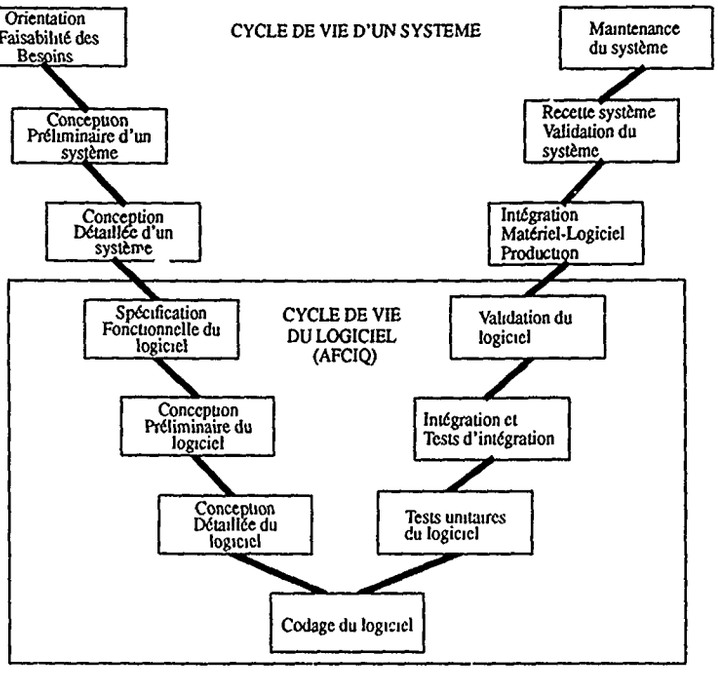


Figure 1 : Cycle de vie d'un logiciel

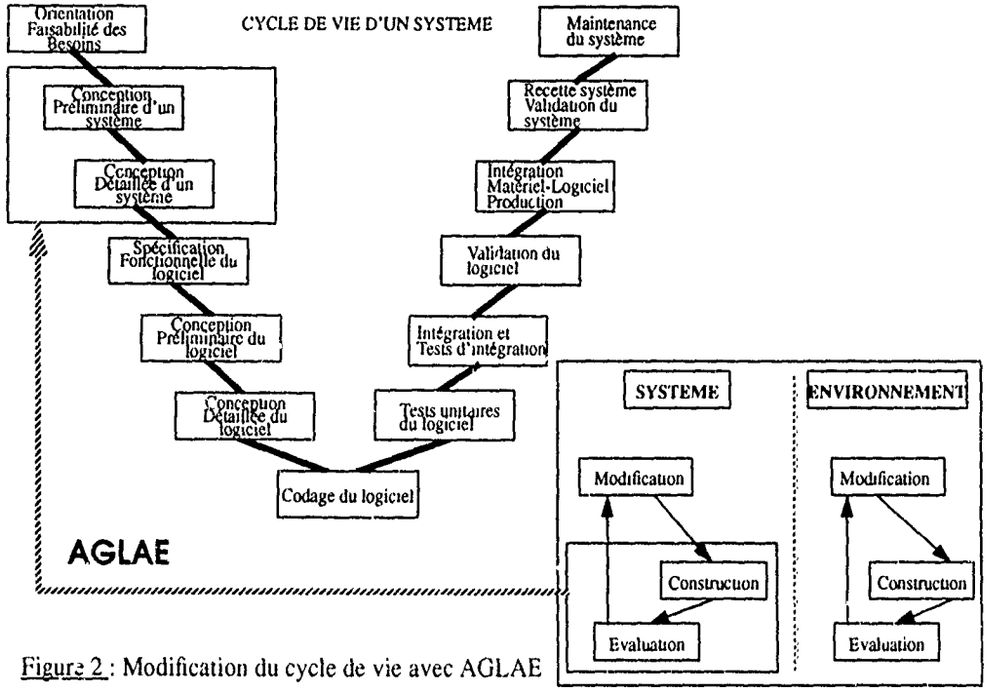


Figure 2 : Modification du cycle de vie avec AGLAE

8 - Planche 2

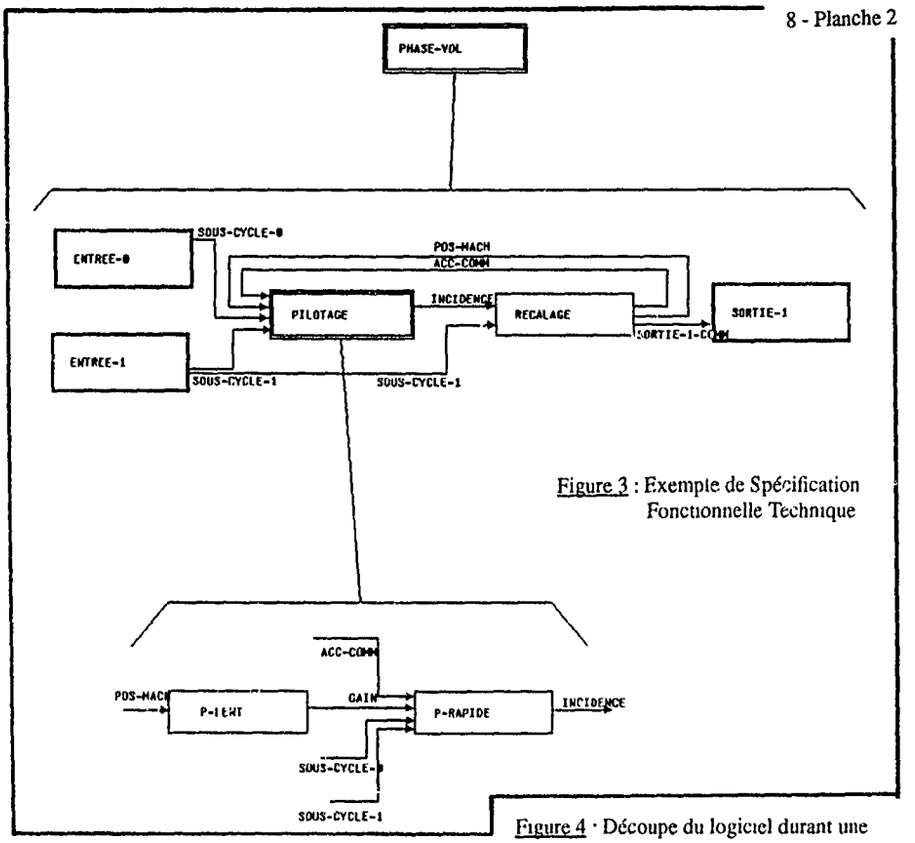


Figure 3 : Exemple de Spécification Fonctionnelle Technique

Figure 4 · Découpe du logiciel durant une période courte

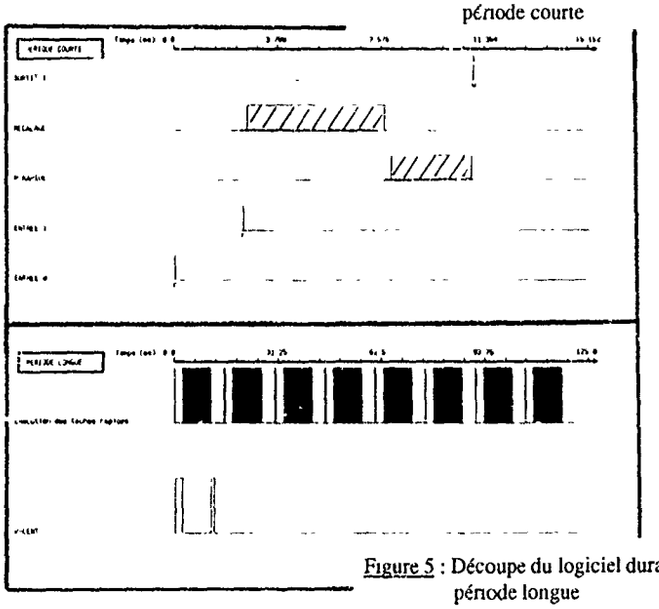
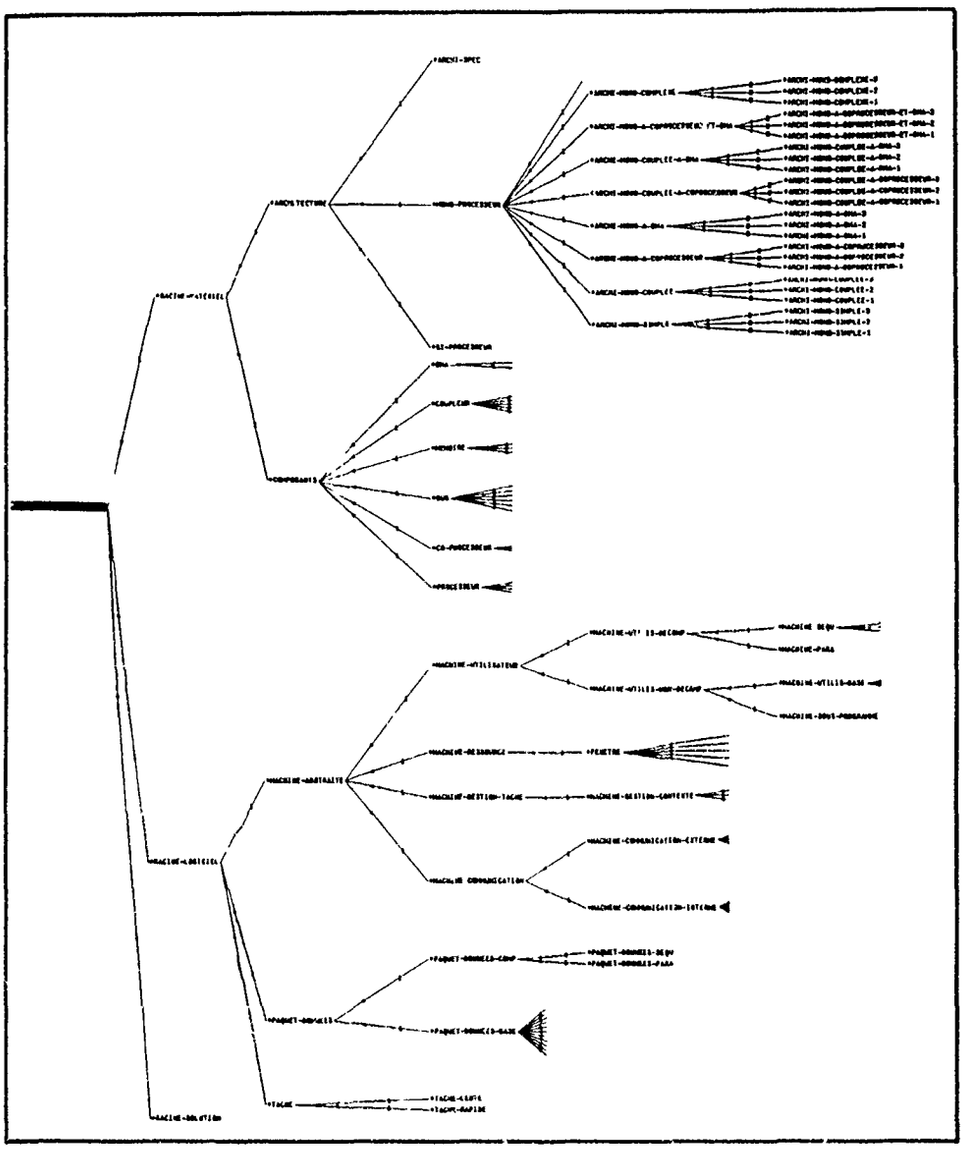


Figure 5 : Découpe du logiciel durant une période longue

Figure 6 : Structure arborescente
de la base de faits



8 - Planche 4

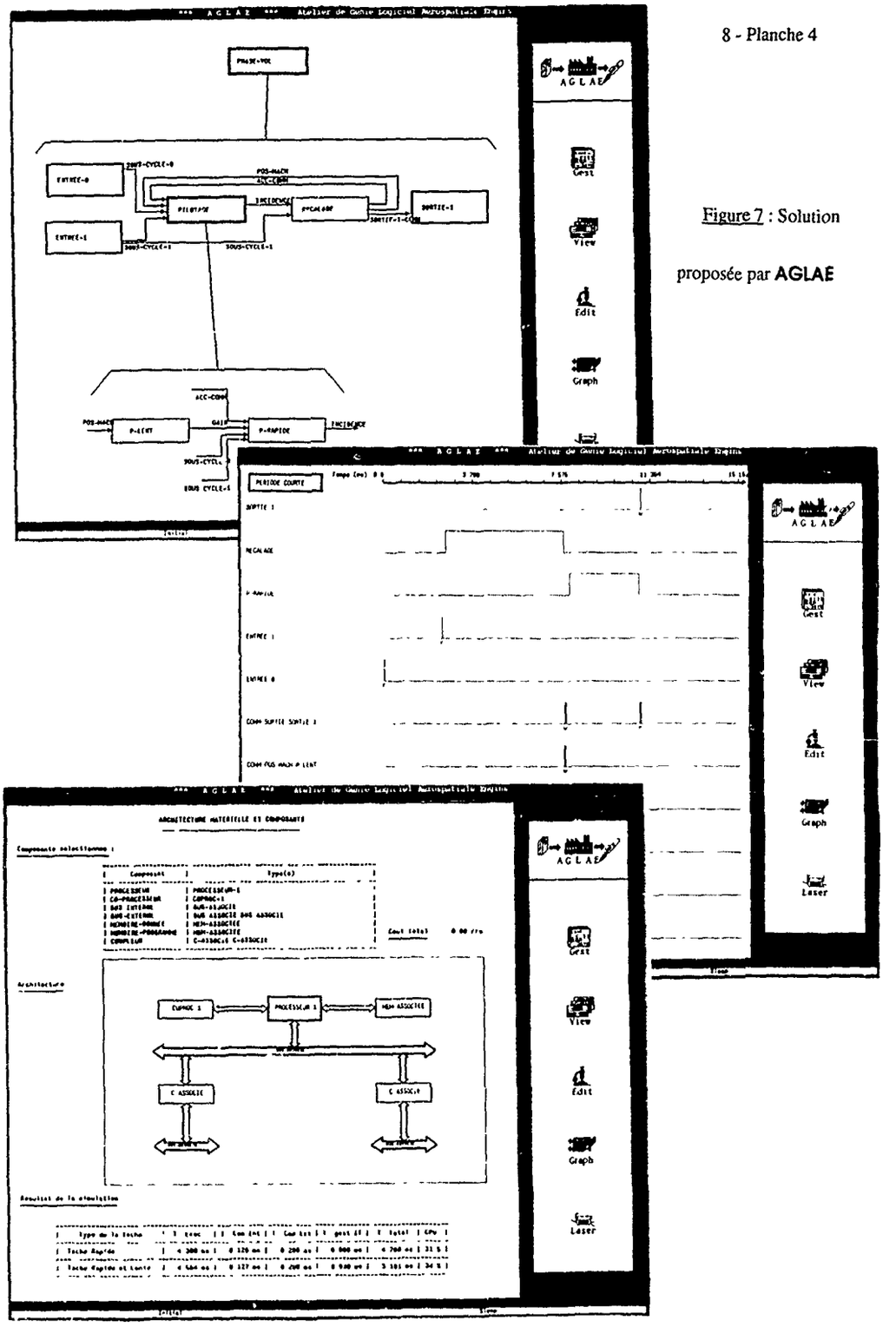


Figure 7 : Solution proposée par **AGLAE**

SOFTWARE DESIGN CONSIDERATIONS FOR AN AIRBORNE COMMAND AND CONTROL WORKSTATION

by
P. Kielhorn, P. Kühl, B. Muth, R. Vissers
Dornier Luftfahrt GmbH
Postfach 13 03
7990 Friedrichshafen 1
Germany

Summary

Within this paper we present some basic concepts of the software design for a command and control workstation for airborne applications. We report not only on theoretical considerations but also on practical experience, which was gained during the development process of a prototype command and control workstation at DORNIER. Special emphasis is put on software archititecture, data structures and tasking with respect to Ada. In order to get a firm basis and to ease understanding the paper starts with a description of the tasks and components of a command and control workstation, which includes a short description of the afore mentioned DORNIER prototype workstation MODOS. The paper concludes with some issues on software "-ilities".

1. Introduction

Airborne command and control (C^2) workstations will be used in a variety of different applications, will range from maritime patrol to border patrol, from military missions to civil missions, and include different flying platforms, i.e. aircrafts as well as helicopters.

A C^2 workstation operates as a link between a human operator and an environment (fig. 1), which normally consists of a suite of sensors and in some cases also of effectors. Basically, employment of a C^2 workstation may be as a stand-alone unit connected to a single sensor, or in a more effective way as a single station controlling multiple sensors/effectors, or within a network of multiple sensors/effectors and workstations, which in addition provides the ability of a redundant design of the system.

2. C^2 Workstation Description

2.1 Capabilities

The following main capabilities have to be provided by a C^2 workstation

- man/machine interface
- sensor/effector management and sensor data acquisition
- data processing
- data storage

These capabilities will be used for applications as for example

- antisubmarine warfare (ASW)
- signal intelligence (SIGINT)
- search and rescue (SAR)
- pollution control
- coast guard
- fishery patrol
- photogrammetry
- verification

Fig. 2 shows the environment for an ASW mission, certainly requiring more than one workstation, whilst the pollution control example in fig. 3 may well manage with a single station.

2.2 Components

From a system point of view a workstation consists of a multitude of parts, from a software point of view however there are only a few components, that have to be mentioned (fig. 4). These components which host the software or work together with the software are as follows

- the workstation processor (main processor),
  the heart of the workstation
- the workstation add-on processors,
  additional or specialized processing power
- the graphic engine with one or more display screens,
  output generation and presentation to the operator
- the operator input devices,
  the means of human input interaction
- the system interfaces,
  the various connections to the outside world

It is obvious that a modular design will ease workstation adaption to special needs.

2.3 MODOS

For the afore mentioned tasks DORNIER have developed a workstation based on a set of requirements of which modularity was one of the most important. Therefore the name of that station was chosen as MODOS standing for MODular Operator Station. The workstation shown in fig. 5 is installed into a Do228 aircraft, which acts as a small maritime patrol aircraft, equipped with a SUPERSEARCHER radar, a KESTREL ESM and a secondary LINS navigation system. The software of this system and the experience gathered during the course of its development is subject and basis of the following design considerations.

3. Software Design Considerations

A command and control workstation is a real-time embedded computer system because of its connection to a real-time process, real-time information gathering and processing, and immediate interaction. Although not extremely demanding, we have to tackle with interrupts, concurrency, multiprocessing and time dependency. A well structured software may help to meet these challenges.

3.1 Architecture

As explained before one of our most stringent requirements was modularity in hardware and software, because modularity is seen as the key to flexibility and adaptability. Modularity should also help to build up an open archtitecture (fig. 6). The first step in this direction is the separation between application dependent software and application independent software. In this way we introduce some kind of structuring, which is of course reasonable, but it is merely suitable for an overview. In order to get a useful architecture, we have to get a deeper insight into the functionality of a C^2 workstation.

Starting with the context diagram of fig. 7a as the first level we achieve by means of usual stepwise decomposition on the second level what we call platforms (fig. 7b) and on the third level what we call building blocks (fig. 7c). A group of building blocks forms a platform. As indicated in fig. 7b we distinguish a man-machine platform, a sensor platform, a data storage platform, and a special processing platform. Fig. 7c gives more detail by showing the building blocks of the man-machine platform, including the operator input/output devices.

3.2 Standards

Another design goal was to rely on standards whenever possible. Where formal standards were not available we applied quasi-standards, i.e. widely used conventions or agreements (table 1).

Table 1: SW-Standards & Quasi-Standards

- Programming Language	Ada
- Operating System	ARTX
- Graphics	GKS
- Symbol Set	national
- Files	IFX
- Data Base	SQL
- Libraries	NAG
- Presentation	X-Window

Ada as a programming language standard is of course the most significant standard for this project. Ada is now a "must" in the military market-place, but the benefits of Ada are also attractive for the civilian area. And because we want to serve both markets we adopted the military standard to the civil product.
ARTX, the real-time executive, is the operating system which is used for our workstation in conjunction with Ada. ARTX is widely accepted in industry, it is a quasi-standard.

GKS stands for Graphical Kernel System, it is one of several graphic standards in use. We use GKS level 2C with standardized Ada language binding.

The symbol set to generate tactical situation presentations is adopted from the German Navy. The symbol set is realized as a data structure and is therefore easily interchangeable.

IFX as an amendment to ARTX is the file handling system. It provides a MS-DOS-like file structure, which is also regarded as a quasi-standard.

We have applied no database standard, no library standard and no presentation standard, mainly because relevant standards were not available when we started our development. A math library standard as NAG (Numeric Algorithm Group) or AGL (Ada Generic Library) was not necessary and not applicable resp. Maybe we will use SQL or X-Window in the future.

3.3 Data strucures

Data flow and data structures are also areas which have to be considered during software design. Whilst data flow is more a global matter to be examined during the analysis phases, data structures have to be determined during the software design steps.

Generally we have to deal with two kinds of data flows

```
o sensor -> processing -> presentation
                            or
                       -> storage

o operator -> processing -> sensor
                              or
                         -> presen-
                            tation
                              or
                         -> storage
```

whereat the second flow is mainly a control flow, only sometimes combined with data.

One of the problems that arise when developing a software structure are the synchronous and asynchronous aspects of the environment, i.e. we have to handle synchronous/asynchronous input and output.

Asynchronous input means that the time when data input occurs cannot be predicted. For example all operator input is asynchronous. Also some sensor types are designed to deliver data only by event, because for embedded computer systems often

interrupts are used to decrease CPU overhead.

Asynchronous output is output of which the amount of generated data is variable. As a result the time needed to generate these data is variable. For example the amount of graphic output data depends on the number of objects to be displayed or refreshed. Output data generated only when a certain event occurs is also asynchronous (e.g. operator action to provide effector data).

Synchronous input is data received with a constant frequency and constant data block size. Most of the sensors with a data bus interface according to Mil-Std-1553B need to be polled within a constant time period to prevent data loss.

Synchronous output is data generated with a fixed time period and block size. For example update/refresh data for intelligent sensors needing actual aircraft position for calculating course or direction is synchronous output.

Data has to be structured in order to be handled properly and effectively. One measure is to create types, but what we want to use is a somewhat higher level of abstraction. This is clarified in fig. 8. This figure shows a section of the input data flow of a distinct sensor, in this case a radar. Incoming data (the messages) is held in different structures according to the relevant processing stage. The structures, which are used here, are called queue and pool resp. Similar to this we have to provide further data structures for other processing purposes. The structures we have used within MODOS are

- fifo for buffering of single data
- queue for buffering of messages
- pool for data base
- tree for describing a hierarchy of commands
- menu for operator command choice
- form for operator generated data entry
- window for data or image presentation

The first 4 structures are of more general character, whilst the last 3 are special for the workstation application.

3.4 Tasks

In an embedded computer system where synchronous and asynchronous data is processed the software design has to guarantee that all events are processed in time independent of the basic processor workload. It is extremely difficult to achieve this within a purely sequential program structure, if there is a number of concurrent processes. One may easily derive concurrency between for example data acquisition, data processing, data presentation, and the various operator inputs. In other words one has to establish a tasking concept.

Our MODOS software design comprises 38 tasks, a fairly high number, and it has to be explained which considerations led to this number. To begin with, these tasks may be sorted into 3 categories, which are shown in table 2.

Table 2: Task classification

Type	Qty	Characteristics
Actor Task	7	cyclic with delay, free-running, no entry initial start synchronized by initialisation routine main purpose to transport data
Manager Task	13	cyclic with delay, free-running, with entries initial start synchronized by initialisation routine main purpose to perform high level system functions
Server Task	18	non-cyclic initial start synchronized by initialisation routine main purpose to provide abstract data types

The first category contains the so called actor tasks. These tasks correspond with all cyclic data transfer, internally as well as externally. The second category, called manager tasks, provides high level system functions. Most of these tasks control the various system modes of the workstation and the environment. The third category is built up by server tasks, which encapsulate a data structure and the associated functions to operate on, in other words an abstract data type.

While the design as a task is judicious for both the first and second category namely concurrency, the decision for the third category has to be explained. An abstract data type may be realized of course by a set of procedures which permit the necessary operations. Now, when different tasks access the same operation, one has to provide some means of protection against intermixing. A task with entries according to the various operations and performing these operations during rendezvous is a very suitable solution.

Tasks of the first and second category are called active tasks. The structure of these tasks may vary, whereat CPU time consumption may be a driving design consideration. Fig. 10a shows the basic active task type, it is not waiting for any external event or rendezvous. Fig. 10b shows a slight

modification to allow for additional communication with other tasks or procedures via entry-calls. Fig. 10c shows a further modification to allow for switch on/off on request. If switched off this task will not consume CPU time until switched on again. The state of the task can be changed by different events (external interrupt, rendezvous, self-deactivation). A last example with respect to task design is given in fig. 10d and shows the combination of a server task and an active task, used for the handling of Mil-Std 1553B data bus. The combination provides for the possibility to use different task priorities. Communication between the two tasks is realized by a fifo buffer.

Designing a task based program structure implies also to take care of deadlocks. Using the full repertoire of Ada language constructs as "select ... or", "select ... else", timed or conditional entry calls, "delay" statement, and watch-dog timers in conjunction with appropriate transfer protocols in the case of multi-processor systems may help to avoid deadlocks. For our MODOS design we did some Petri-net modeling, but because of deficiencies of the tool used, we finally relied on more empirical methods.

3.5 Interfaces

A C^2 workstation is part of a more comprehensive system, i.e. there are interfaces to the environment, which of course should also be based on standards (table 3).

Table 3: Interface Standards

Hardware:	STANAG 3838 (Mil-Std 1553B)
	RS232C/RS422
	SCSI
	RS 485
	ARINC 429
	others
Software:	GKS Level 2C
	SQL
	others

The first kind of interfaces is the connection to the external environment and to the workstation's peripheral devices. From a software point of view the only consideration is whether the necessary drivers have to be written in high order language or in assembly language. The next consideration refers to the man-machine interface. But this interface, although software-driven, depends on system considerations. The third kind of interface is the software interface. Relevant standards we have mentioned in the beginning. Beside these one has to pay attention to the required open architecture. The software structure should be designed in such a way, that new applications, i.e. new software modules may be built in easily.

4. Assessment

We would like to pick up again some of the before addressed aspects, in order to assess their relevance within the MODOS workstation software design. These are

- Open architecture/modularity
- Reusability
- Portability
- Testability/maintainability
- Real time processing capability/ tasking

In addition also some aspects of object oriented design (OOD) will be discussed.

4.1 Open architecture/modularity

An open architecture shall allow an ease expansion by new modules to obtain additional features. As shown before the hardware is designed for add-on processors thus giving the capability of a separate database processor, an extra signal processor or something like that. This implies the distribution of software. One of the expected expansions may be the employment of a new sensor. In this case not only the addition of new modules is necessary, but also changes in the existing software have to be made. As long as data structures as menus or forms are concerned, they may easily exchanged. The incorporation of new commands into the control task is shown in fig. 9. Modularity gives us still more flexibility: exchangeable symbol sets, exchangeable maps, variable function key assignment for example.

4.2 Reusability

Modularity and exchangeable data structures promote also reuseability. For new applications almost all of the basic software represented by the inner shells of the formerly shown software layers diagram (fig. 6) may be reused. All abstract data types are candidates for reuse. For example within MODOS there are 14 instances of the abstract data type "queue".

4.3 Portability

Portability is widely supported, because only a small amount of code, less than 5 %, is written in assembly language. With one exception no services of the underlying operating system are used directly by the software. Of course a new environment has to provide a GKS interface.

4.4 Testability/Maintainability

Both "-ilities" are strongly supported by modularisation and standardisation. Furthermore a clear architecture and identical or similar instances of software components assist in testability and maintainability resp. That means, that both will be achieved automatically at least partly if the basic design considerations are obeyed

(besides such means as comments, programming style etc.). Of course testability will be complicated by concurrency.

4.5 Real time processing capability/ tasking

The extensive use of tasks may cause problems with real time processing because of accumulated task switching time overhead. We have experienced no detrimental effects, which is of course owning to the ARTX operating system. The Do228 MPA example mentioned at the beginning shows a balanced behaviour between processing capacity and data transport capacity. Furthermore the tasking concept is helpfull when distributing software in multiprocessor systems.

4.6 OOD

Because object oriented design (OOD) is now the favorite software design method, we are caused to make some remarks on it.

For the software design of MODOS no special OOD method and tool were applied. Nevertheless the software architecture was built up in accordance with the OOD philosophy of Grady Booch. Furthermore the application of the Ada language supports the implementation of OOD by providing of implementation features as follows

- o structured constructs
- o typing
- o interface specifications
- o information hiding and data abstraction

The MODOS software architecture is implemented by an Ada package hierarchy which is defined by the use relationship. The objects of the MODOS software are represented by Ada packages or compositions of Ada packages.

The top level objects like (see also fig. 7)

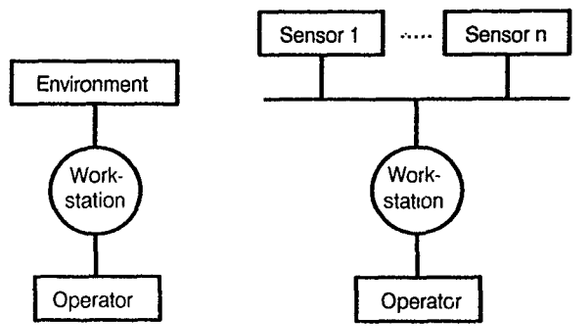
- o Environment I/F Management
- o Sensor Management
- o Data Storage
- o Data Fusion/Tactics
- o MMI

are created in accordance with the real-world objects of the MODOS system. Those objects group only lower level objects to one object which represents a real-world entity.

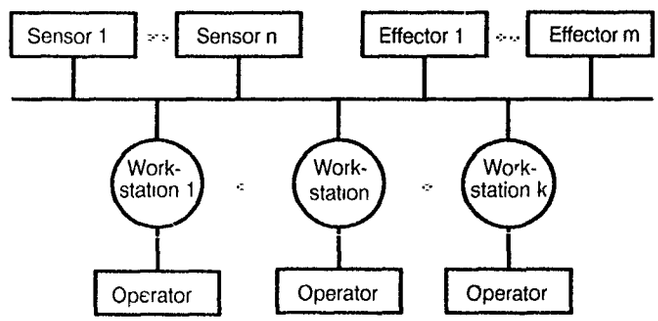
From the software engineering point of view more efficient objects are the low level objects of the MODOS software architecture. These objects are implemented as an Abstract Data Type. Most of them are multiple used.

5. Conclusion

Some considerations concerning software design of a command and control workstation have been presented and their realisation explained and discussed. The requirements for a C^2 workstation lie above all in adaptability to different kinds of employment. This is achieved by the current software design. Far-reaching requirements as hard real time processing or extreme data throughput may be satisfied by add-ons to the basic design. Because of the principles of modularisation and standardisation for software as well as for hardware stronger requirements will easily be met.



a) Basic Configuration b) Multiple Sensor - Single Workstation Config



c) Multiple Sensor/Effector - Multiple Workstation Configuration

Fig. 1: System Configurations

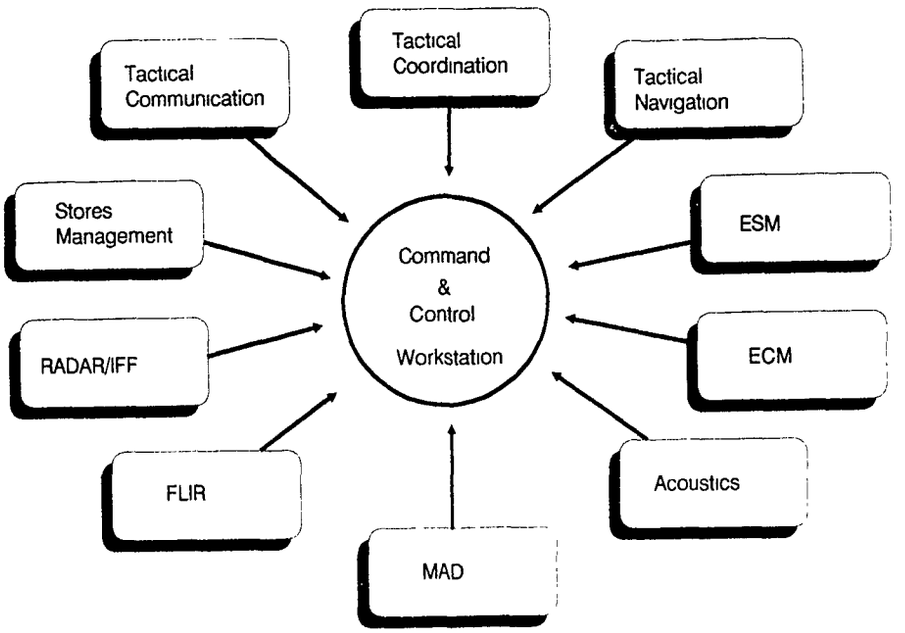


Fig. 2: C^2Workstations Applications in ASW Missions

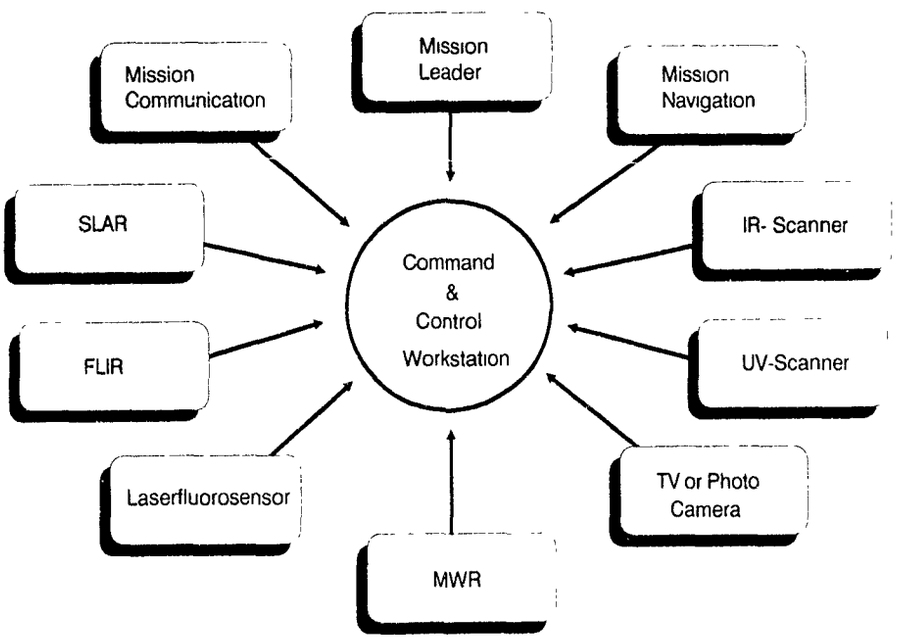


Fig. 3: C^2Workstations in Oil Spill Detection Missions

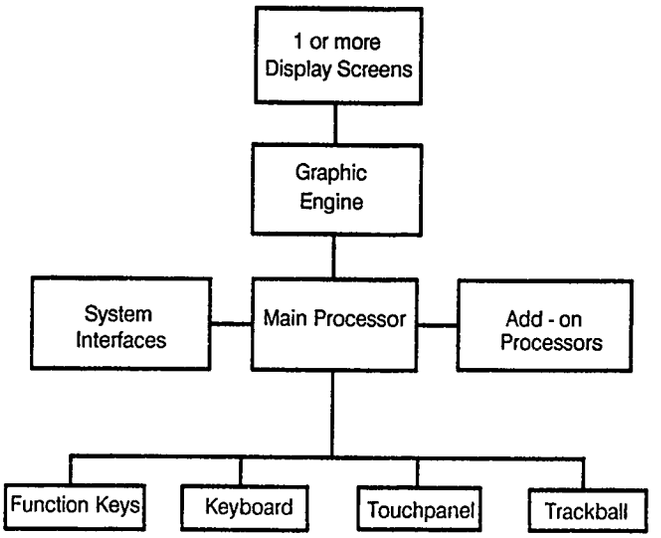


Fig. 4: C^2 Workstation Blockdiagram

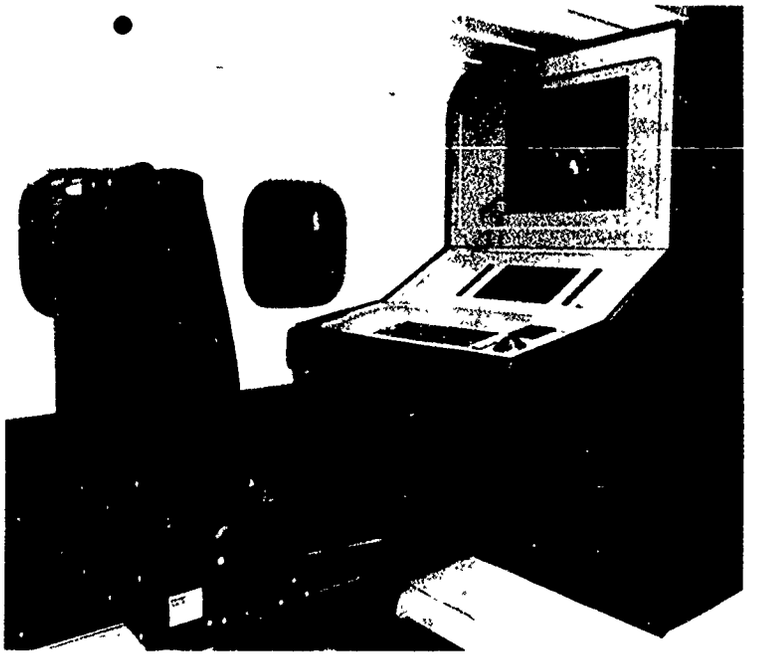


Fig. 5: Modular Operator Station MODOS

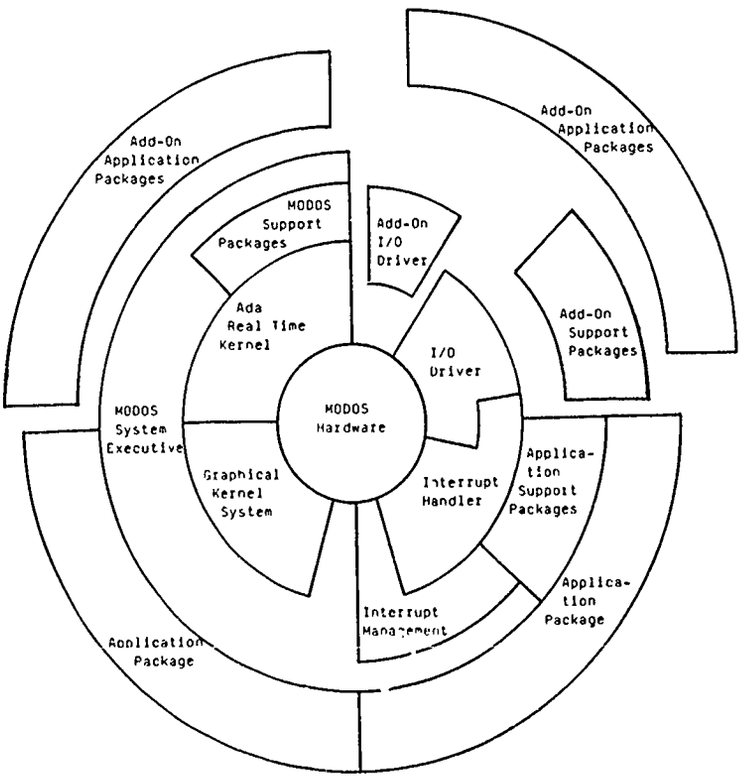
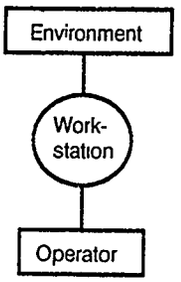
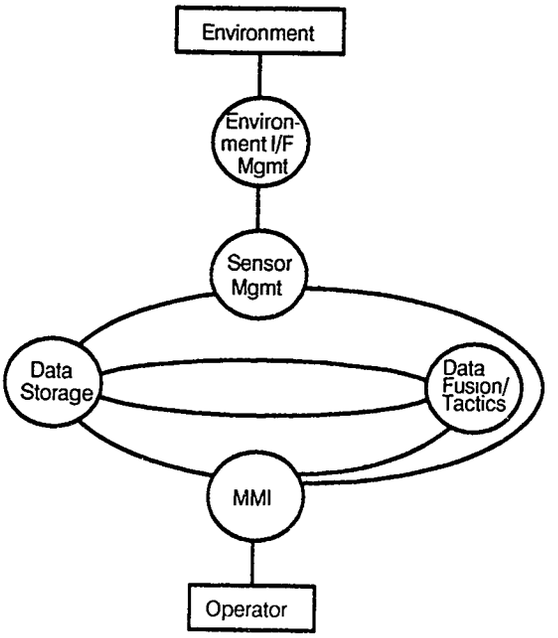


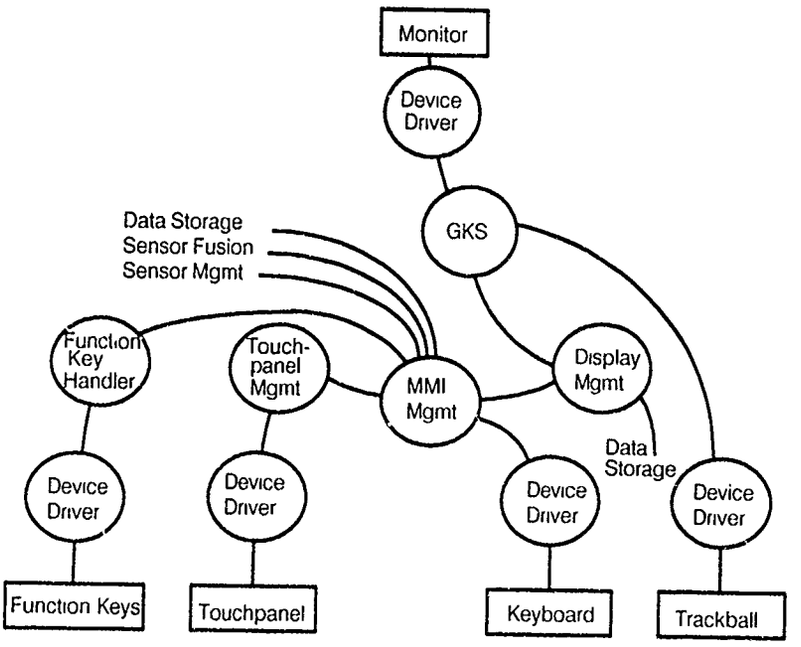
Fig. 6: Software Layers



a) Context Diagram



b) First level of decomposition



c) Second Level (first level of decomposition of MMI)

Fig. 7: SW Architecture - Decomposition

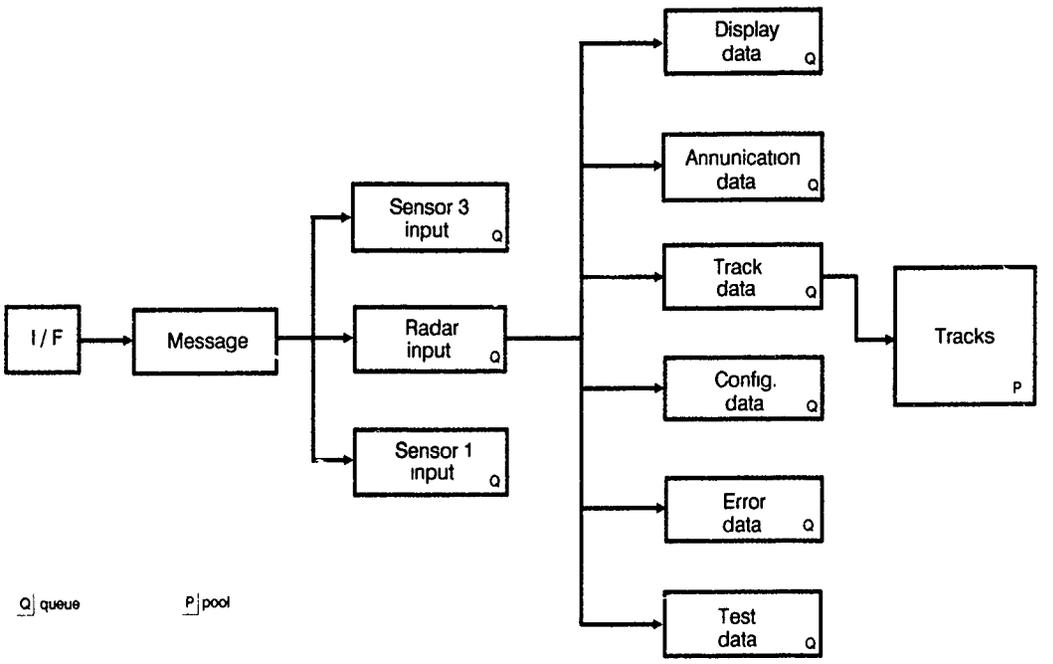


Fig. 8: Data Flow

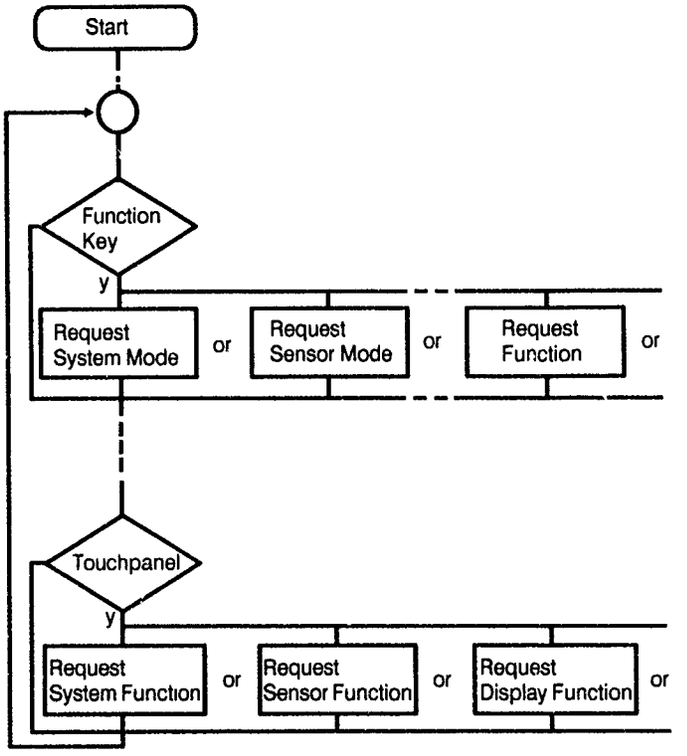


Fig. 9: Principle of mode/function control

```
task body actor_task is
--
-- local declarations
--
begin
   --
   -- Before the task can run it must be
   -- explicitly started.
   --
   accept start (...) do
      --
      -- initialize variables and return
      -- initialisation status.
      --
   end start;
   --
   -- Now the task is active and can run freely
   -- except when the delay-statement
   -- is executed.
   --
   loop
      statements;
      --
      -- allow other tasks to run
      --
      delay specific_time;
      statements;
   end loop;
end normal_task_type_1
```

Fig. 10: Active Tasks Examples
a: free-running with delay

```
task body manager_task is
--
-- local declarations
--
begin
   --
   -- Before the task can run it must be
   -- explicitly started.
   --
   accept start (...) do
      --
      -- initialize variables and return
      -- initialisation status.
      --
   end start;
   --
   -- Now the task is active and can run freely
   -- except when the delay-statement
   -- is executed.
   --
   loop
      select
         accept entry_a (...) do
            -- perform rendezvous
         end entry_a;
         [ statements; ]
      or
         accept entry_b (...) do
            -- perform rendezvous
         end entry_b;
         [ statements; ]
      or
         [ when boolean_expression => ]
         accept entry_c (...) do
            -- perform rendezvous
         end entry_c;
         [ statements; ]
      .
      .
      .
      else
         [ statements; ]
         --
         -- allow other tasks to run
         --
         delay specific_time;
      end select;
   end loop;
end normal_task_type_2
```

Fig. 10 (cont.): Active Tasks Examples
b: free-running with delay and entries

```
task body switching_task is
--
-- local declarations
--
begin
   --
   -- Before the task can run it must be
   -- explicitly started.
   --
   accept start (...) do
      --
      -- initialize variables and return
      -- initialisation status.
      --
   end start;

   inactive:
   loop
      accept activate_entry (...) do
         -- perform rendezvous
      end activate_entry;
      active:
      loop
         select
            accept deactivate_entry (...) do
               -- perform rendezvous
            end deactivate_entry;
            [ statements; ]
         or
            accept entry_b (...) do
               -- perform rendezvous
            end entry_b;
            [ statements; ]
         or
            [ when boolean_expression => ]
            accept entry_c (...) do
               -- perform rendezvous
            end entry_c;
            [ statements; ]
         .
         .
         .
         else
            [ statements; ]
            --
            -- possibility for the task to
            -- deactivate itself
            --
            exit active when ....;
            --
            -- allow other tasks to run
            --
            delay specific_time;
         end select;
      end loop active;
   end loop inactive;
end switching task;
```

Fig. 10 (cont.): Active Tasks Examples
c: same as 10b plus switching capability

```
package body milbus is

   task body manager is
   -- this task has the default task priority

      task driver is
         pragma priority (system.priority'last);
         -- highest priority

         entry start (frame_time : in duration);
         entry milbus_interrupt;
            for milbus_interrupt use at ...;
      end driver;

      task body driver is
      fr_time : duration;
      begin
         accept start (frame_time : in duration) do
            -- perform rendezvous
            fr_time := frame_time;
            [ statements; ]
         end start;
         loop
            [ statements; ]
            accept milbus_interrupt;
            [ statements; ]
            delay fr_time;
         end loop;
      end driver;

   begin -- of milbus.manager
      accept phase_0 (...) do
         [ statements; ]
      end phase_0;
      --
      -- begin of phase 0
      --
      phase0:
      loop
         select
            accept enter_message (...) do
               [ statements; ]
            and enter_message;
         or
            accept phase_1;
            exit phase0;
         end select;
      end loop phase0;
      --
      -- begin of phase 1
      --
      phase1:
      loop
         select
            accept new_frame (...) do
               [ statements; ]
            end new_frame;
```

(to be continued)

```
         or
            accept enter_message_descriptor (...) do
               [ statements; ]
            end enter_message_descriptor;
         or
            accept phase_2
               (frame_time : in duration ) do
               [ statements; ]
            end phase_2;
            exit phase1;
         end select;
      end loop phase1;
      --
      -- begin of phase 2
      --
      driver.start (frame_time);
      --
      -- now the driver task is activated
      --
      phase2:
      loop
         select
            accept available_messages
               (count : out natural) do
               [ statements; ]
            end available_messages;
         or
            accept get_message (msg : out message) do
               [ statements; ]
            end get_message;
         or
            accept send_extra_message
               (msg : in message) do
               [ statements; ]
            end send_extra_message;
         or
            accept change_message (...) do
               [ statements; ]
            end change_message;
         end select;
      end loop phase2;
   end manager;
end milbus;
```

Fig. 10 (concl.): Active Tasks Examples
d: Combination of active task and passive task

Formal Specification of Satellite Telemetry: a Practical Experience*

Jean-Michel HUFFLEN
GRECO INFORMATIQUE

Michel LEMOINE
ONERA-CERT/DERI

2, avenue Édouard-Belin
31055 TOULOUSE CEDEX
FRANCE

Abstract

We expose an experience of using formal algebraic specifications, conducted in collaboration with an aeronautic industry. The objective is to provide a reusable specification of processing telemetry results. This family of spatial applications is described by means of generic formal specifications, and each telemetry could be built from them. Reuse possibilities are supported by our framework In this paper, we give a general survey of this experience, including its "story", the method followed for establishing the generic specifications which are the system core, and reuse aspects provided.

Keywords: formal specification, telemetry decommutation, design and software reuse, fast prototyping, requirements elaboration.

Introduction

It is obvious to say that software becomes more and more complex and of course as a direct consequence more and more expensive to develop. Too much work, too much time are necessary to get an operational version of a software product. Among the reasons responsible of such difficulties in the software production, at least two are of first importance.

First of all, the development process activity is slowed down each time the description of the system to be produced is not precise enough and requires to make decisions all along development. Unfortunately, this appears any time the requirements document is expressed in a natural language. It is impossible to guarantee that such a document describes, in an unambiguous way, what the system has to do. How to know whether or not the document is *complete* (has everything been said?) and *consistent* (is there no contradiction)?

*This work was supported by a contract between CNES and GRECO-PRC PROGRAMMATION. This action of technology transfer has been managed by two teams in Grenoble (IMAG/LIFIA) and Toulouse (ONERA-CERT/DERI).

The second reason is the origin of the *reuse* notion. Many studies have shown that a lot of functionalities are repeated from one application to another. What about reusing at least final codes? It is clear that the ability to reuse software is a main key to the software success: the gains we can expect are of course important from the viewpoint of economy in terms of work and time But another perhaps more important gain is obtained if the reused and/or adapted parts have been proved correct once and for ever

A means to decrease the underlying difficulties is to start the development process—before beginning the coding phase—by a first description of the application in a very high level formalism. The goal of this description we call *specification* is to express the application in a *formal* language. By *formal* is meant a language for which the mathematical foundations are precisely established both from the viewpoint of its syntax and semantics. Because the use of a formal language requires a *precise* description of all the wished functionalities, any problem expressed with such a formalism will have *one* and *only one* interpretation, the same for all kind of users (men and computers).

Among the precise specification formalisms, one seems quite interesting. it is called *algebraic* specification because it is based on specifying types as value sets and functions in an equational way to express function behaviour. Moreover, it is possible to *parameterize* an algebraic specification by another which represents *hypotheses* Such parameterized specifications are also called *generic* specifications At the last, let us remark that under certain conventions, some algebraic specifications can be run and play the role of a *prototype*. (Concerning this specification formalism, its theoretical notions can be found in [4].)

In all the cases, the specification activity obliges to ask ourselves the right questions and to give the right answers by fixing—in a not necessarily definitive way—the choices related to the functionalities to provide. This step is compatible

with the reuse problem. Indeed when we consider related applications, the use of *abstraction* for synonymous functionalities allows a better obviousness of common parts.

In this paper, we present an experience of using formal specifications within the context of an industrial and spatial application: **telemetry systems**. As it will be mentioned later, the reuse objective is always present. A complete description of this experience is available in [9]. [10] points out the didactic results of this operation and also gives the CNES (Centre National d'Études Spatiales: National French Space Agency) viewpoint. In this paper, we emphasize more the followed process and the main results considered from a methodological viewpoint. In Section 1, we quickly recall the study evolution. Section 2 points out our principles for writing this specification. Section 3 summarizes the lessons and perspectives this work opens.

1 "Story" of the study—The chosen approach

The telecommunication division of the CNES is in charge (among others) of writing the requirement documents relative to the telemetry systems. Then it asks software houses to develop the corresponding software system. Up to now, all the telemetry systems were developed independently each others even if obviously some phases of the telemetry processing are similar from one telemetry to another. Here is the starting point of our study: to show that some (supposed) reusable parts can be effectively reused for any kind of telemetry systems by means of reusable software components.

Our basic idea was to develop an *only* and *formal* requirement document from the informal ones which described a few existing telemetry systems. Instead of this, another approach should have been to develop more or less directly reusable software components. This second approach has not been achieved for *assessment* reasons and also to guarantee that the reusable software can be keep free from any implementation language.

The study was started with five different telemetry systems. We tried to exhibit all the common functionalities between them. This top down approach was abandoned because.

- the difficulty to understand the informal documents. indeed, there are so many incompletenesses, too much implicit that only a specialist of this domain was able to understand such documents;
- each document was specific to a given telemetry system: in other words, each document reflects the description of one telemetry system outside any global consideration—for instance, the used terminology was often different from one system to another.

It might have been possible to isolate in a very hand fashion the common functionalities but without any warranty of keeping consistent the overall understanding of what a telemetry system was. Another point to be remember is the fact that the documents we had in hands were about *yet existing* telemetry systems and that we were interested in developing a general and formal telemetry description for *future* systems. What should have been the adequacy of a new system in such a context? What credibility should have been given? This problem has been mentioned many times: *what has not been developed with a reusability aspect is difficult, even impossible to reuse!* Here was the main reason we started in another direction.

First of all we asked the telemetry specialists to describe in an informal but rigorous manner what a telemetry system is from an abstract viewpoint. Then after several inspections, this document [5] was formalized in terms of *generic* specifications we will call **generic telemetry** in the rest of the paper. Of course, all the problems were not solved in [5]. Nevertheless, its main advantage was that it was independent on any existing or future telemetry system and it was readable enough for non specialists of the field as we were.

The generic telemetry gathers the description of all the entities involved in any telemetry system. It looks like a general framework from which any telemetry specification could be built. It is evident that the informal general document was not able to take into account all the cases. Thus the generic telemetry is open (e.g. new elements can be easily added) and very abstract in order not to be too restrictive or too dependent on any particular case. In practice to the contrary of the first approach, this second one is definitively bottom up. Indeed, it is more simple to *enrich* a generic specification with new elements (not yet considered as generic enough) than to *suppress* peculiar characteristics that had unfortunately been considered general.

Having written the generic telemetry during the first part of this operation, it was interesting to test it on a real case. We have done this work with a telemetry of the scientific project INTERBALL. As mentioned above, it was necessary to enrich the generic telemetry. The example has been fully developed for its main difficult parts: generation and storage of embedded information according to an INTERBALL format, and decommutation of the received data at ground.

A few remarks:

- due to the chosen specification language we have been able to run parts of this instantiated specification. This point is very comfortable.

Indeed, it is not necessary to wait for the end of the project to get some results allowing the specifier to prove the project under consideration is really effective;

- the internal and external presentations of information are similar: this improves the readability of the formal specification and allows updating it easily in case of any modification to the external view;

- the enrichment of an instance of the generic telemetry must be done with in the mind the reuse goal of the added information (we will go thoroughly this point in §3.1).

2 Specifying telemetries: tool and method

Now we are going to tackle more technical features. First, we describe any telemetry process succinctly. Then we explain how we have specified is as the generic telemetry using generic specifications. We illustrate our method by an excerpt from these specifications, from which we take some examples At the last, we briefly show an example of its use

2.1 General description of telemetries

The aim of any telemetry process is to make up the measurement results effected by a scientific satellite The data of each experiment must be provided in chronological order to the organization which manages it.

Any telemetry system is divided in two subsystems: an *on-board* system and a *ground* one, as shown in Figure 1. The on-board system includes a *transmitter* which sends telemetry results to ground stations. If the satellite is geostationary, transmissions are direct and purely sequential. In the general case, the satellite moves. There are several stations, and the satellite faces one of them only during a little while. While the satellite does not face any station, it records telemetry results to be transmitted in a storage zone. When it is within sight of one, it transmits both direct and recorded telemetries. That is why it is necessary to order results according to their dates afterwards.

Information provided inside satellites is constituted by *bit strings*, according to a specific format for each embedded experiment Before transmission to ground, the satellite mixes these bit strings into *byte matrices* according to a distribution called **telemetry format**. Such a format depends on the considered satellite. On the ground, separating data according to their origin among the received byte matrices, in order to reconstruct original bit strings, is called **telemetry decommutation.**

2.2 Our approach

2.2.1 Frame

As we have exposed in Section 1, our approach proceeds by using parameterized specifications. For any telemetry, our framework for specifying is divided in two parts:

- the **generic telemetry specification:** it includes, on the one hand, the operations which are present in all the telemetries (e.g. a sort according to a time base), on the other hand, the description of all the entities which participate in any telemetry and functionalities which equip these entities.

- the **instantiation,** i.e. the bindings of *formal parameters* to *actual values.* These actual values are supposed to be specified: they represent the specific conventions for the considered telemetry.

In order that modularity of our specification respects the telemetry organization given in Figure 1, we have written one module for one entity. These modules are parameter specifications: because of their generalness character, they do not represent the entities of a particular telemetry, but they must be *templates,* such that the specification of any telemetry entities can be obtained by replacing the module parameters by the conventions of this telemetry.

2.2.2 A limitation

For any telemetry, the number *nb-exp* of embedded experiments and *nb-ts* of technologic sets (cf. Figure 1) depends on the considered telemetry. Let us recall that any experiment and technologic set are represented by a parameter specification in our framework, according to the principles we have stated in the previous subsection. If we consider any telemetry process in a global way, we obtain parameter specifications which are parameterized *themselves*--they are parameterized respectively by the natural numbers *nb-exp* and *nb-ts*.

As far as we know, any algebraic specification language provides neither theoretical nor practical tools to describe such a specification. Some ways to cope this limitation exist. Since we are interested in providing a very readable specification especially, we have chosen the solution we think it is the simplest in a didactic way: we consider *one* experiment, and *one* technologic set. Generalizing these two specifications to describing *nb-exp* xperiments and *nb-ts* would make them more complex, but does not constitute a real problem. In order to be exhaustive, it seems important to us to report this limitation but we *claim* it is the *only* actual limitation we come up against while specifying these spatial applications.

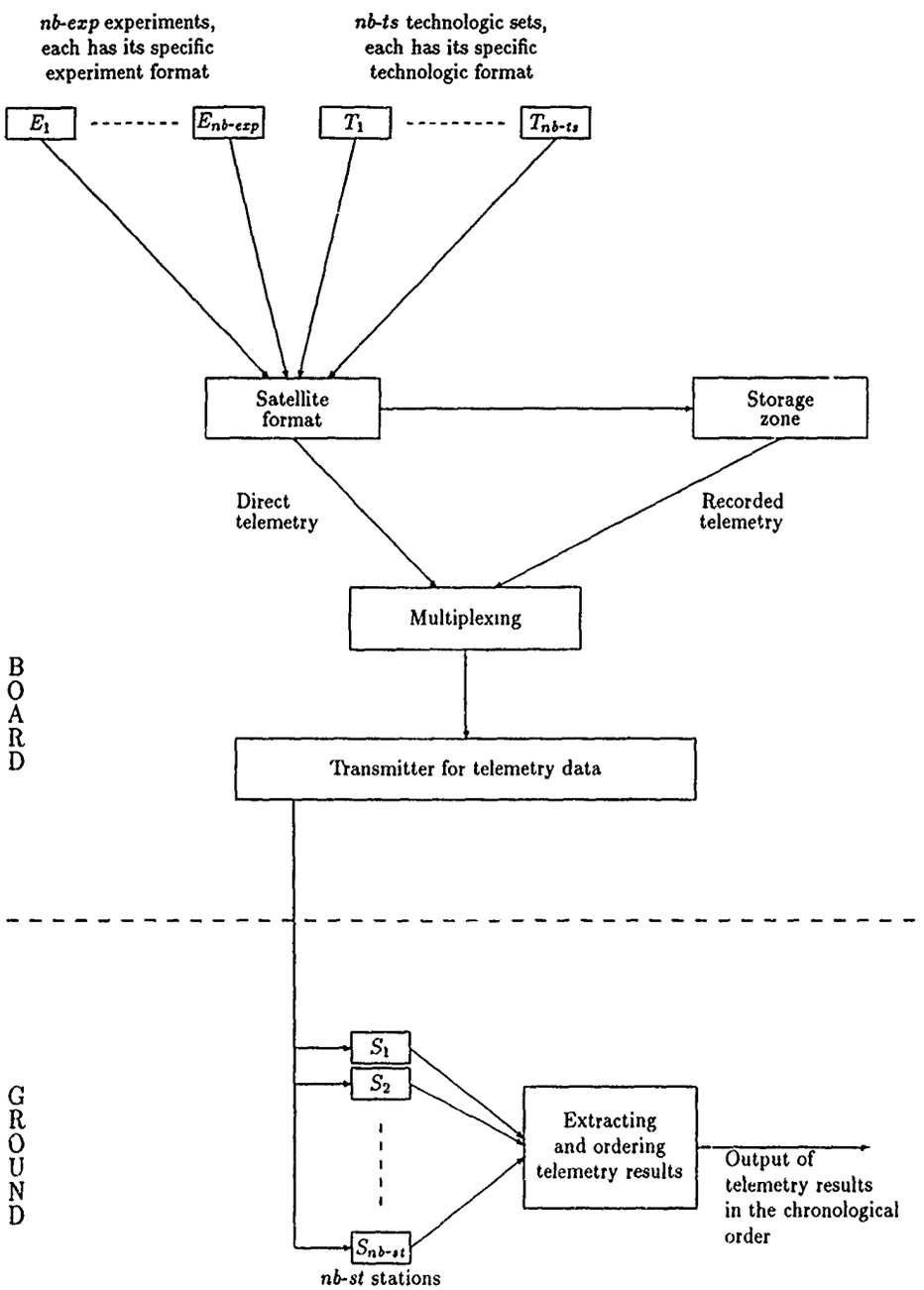


Figure 1: Advancing data during any telemetry (according to [5]).

2.2.3 Method

Now we are going to explain the principles which have guided us in writing the modules of the generic telemetry specification Figure 2 gives one of representative excerpts. Readers interested in the complete text can find it in [8] Some technical notions about our notations are given in Annex.

Because of their algebraic character, the specifications we propose comprise both *types* and *functionalities*. In these parameter specifications, we understand types as *possible value sets*: for example, the modes of a telemetry, which give access to the different telemetry formats, form a specific *type* for this telemetry (In Figure 2, we note *tm-mode* this type which depends on the considered telemetry)

Functionalities are *actions* on types: they are characterized by their inputs and outputs, and they are represented by means of operators For example, reception of data from the experiment is made by means of the operator

$$receive\text{-}exp$$

whose domain is·

$$\begin{array}{l} tm\text{-}mode \\ \quad \times \\ Seq[Fixed\text{-}Array\ Ftype\text{-}and\text{-}Nat[Bit\ /\ exp\text{-}dim_1]] \\ \quad \times \\ Seq[Matrix\ Ftype\text{-}and\text{-}2\text{-}Nat[Byte \\ \qquad\qquad\qquad\qquad / \\ \qquad\qquad\qquad\qquad nb\text{-}rows, nb\text{-}bytes]] \end{array}$$

and codomain is.

$$\begin{array}{l} Seq[Matrix\ Ftype\text{-}and\text{-}2\text{-}Nat[Byte \\ \qquad\qquad\qquad\qquad / \\ \qquad\qquad\qquad\qquad nb\text{-}rows, nb\text{-}bytes]] \end{array}$$

(For more details about these type expressions and meaning of our identifiers, see the commentaries in Figure 2 and Annex.)

In a concrete way, the argument of sort *tm-mode* is the current working mode when the data are received These data are organized in an array sequence using the experiment format, and this operation takes also a matrix sequence using the telemetry format—obtained from *tm-mode*—in which information is put in the appropriate places In another way, the "new" matrices are obtained from the "old" ones by addition of experiment data (Let us recall that there *cannot* be side effect since we consider *functions* in a purely mathematical sense)

According to [5], a telemetry format is a byte matrix whose dimensions are specific to the considered telemetry. In such a way, the distribution inside this matrix depends on the telemetry, too. And *no* additional information is provided. Principles for these distributions may be *very different* from one telemetry to another. Let us cite two examples here—cf. [8] for more details.

The formats of the UARS-WINDII telemetry is divided in vertical sections corresponding to the different kinds of data which they contain As another way, the INTERBALL telemetry formats are described byte by byte

Since no unified approach for describing any telemetry format exists, our solution is to consider a telemetry format as an object, according to the object-oriented approach. We do not know its representation and we have access to it only by *methods*. In our case, the methods are:

- **storing** data into telemetry formats—it is done by means of:
  - ⋆ the operator *receive-exp* for the telemetry experiment.
  - ⋆ the operator *receive-techno* for the technologic set
- **decommutation** of formats in order to provide the different kinds of data these two operators (for the experiment of the telemetry and its technologic set) are included in the specification of ground operations.
- some additional information bound to the format—e g. the synchronization words· they are used to control the reliability of the data flows

This point ends the presentation of the principles we have followed for exhibiting types and operators from the informal document [5]. We cannot go thoroughly the specification of storing and decommutation since it narrowly depends on the considered telemetry format. Thus our generic telemetry is a framework for formally specifying how to built any telemetry specification from it Applications to reuse objective will be seen in §3 1, after a short survey about the instantiation we have studied during the second part of this work

2.3 Instantiating the generic telemetry

As mentioned already, the distribution of the INTERBALL telemetry formats is given byte by byte, according to the different kinds of data. An important point for our specification is that our way for describing the format conventions is very near to what is depicted in the project documents. For storing and decommutation, the different bytes are used as a *grid* in which holes indicate places for deposing or extracting information [9]. We do not detail this feature here, and are going to be rather interested in replacing some elements of the generic telemetry For example, some dimensions are

-- This is a commentary. See Annex for more details about some technical notions.

prop *Satellite-Features*[*tm-mode*, -- Working mode: it provides access to each
-- telemetry format used.
bit-rate -- Type of all different bit rates for
-- transmissions.
/
-- How to receive the results...
$exp\text{-}dim_1$, *receive-exp*, -- ...of *one* experiment (cf. §2.2.2) whose format
-- dimension is $exp\text{-}dim_1$,
$techno\text{-}dim_1$, *receive-techno*, -- ...of *one* technologic set whose format
-- dimension is $techno\text{-}dim_1$.
-- In these two cases, let us recall that the used
-- telemetry format depends on the working
-- mode *tm-mode*.
nb-rows, *nb-bytes*, -- Dimensions for the matrices of telemetry
-- formats.
dim_1, dim_2, *corr*, -- Data enrichment by an error-correcting code.
-- dim_1 and dim_2 are respectively the
-- dimensions of the original array and the
-- enriched one. *corr* is the coding operator.
mode-rate, -- Bindings: working mode $\mapsto$ corresponding bit
-- rate.
obtain-time, -- How to get dates.
lg-synchro, -- Common length of synchronization words.
synchro] -- The synchronization words themselves.

opns -- Domains and codomains of the operators:

$exp\text{-}dim_1, techno\text{-}dim_1, nb\text{-}rows, nb\text{-}bytes, dim_1, dim_2, lg\text{-}synchro$: – $\rightarrow$ *Nat*

receive-exp : *tm-mode* × *Seq*[*Fixed-Array.Ftype-and-Nat*[*Bit* / $exp\text{-}dim_1$]] × *Seq*[*Matrix.Ftype-and-2-Nat*[*Byte* / *nb-rows*, *nb-bytes*]] $\rightarrow$ *Seq*[*Matrix Ftype-and-2-Nat*[*Byte* / *nb-rows*, *nb-bytes*]]

receive-techno : *tm-mode* × *Seq*[*Fixed-Array.Ftype-and-Nat*[*Bit* / $techno\text{-}dim_1$]] × *Seq*[*Matrix.Ftype-and-2-Nat*[*Byte* / *nb-rows*, *nb-bytes*]]) $\rightarrow$ *Seq*[*Matrix.Ftype-and-2-Nat*[*Byte* / *nb-rows*, *nb-bytes*]]

corr : *Fixed-Array.Ftype-and-Nat*[*Bit* / dim_1] $\rightarrow$ *Fixed-Array.Ftype-and-Nat*[*Bit* / dim_2]

mode-rate : *tm-mode* $\rightarrow$ *bit-rate*

obtain-time : *Nat* $\rightarrow$ *Time*

synchro : – $\rightarrow$ *Seq*[*Fixed-Array.Ftype-and-Nat*[*Byte* / *lg-synchro*]]

-- No specific axiom in these module (cf. §2.2.3).

includes -- Using the specification of *any* error-correcting code:
Using-Error-Codes[/ dim_1, dim_2, *corr*],
-- General binding of modes to rates:
Bit-Rates-and-Modes[*tm-mode*, *bit-rate* / *mode-rate*]

endprop

Figure 2: Data reception inside any scientific satellite.

instantiated as follows:

$$\begin{aligned} \textit{nb-rows} &\mapsto 32 \\ \textit{nb-bytes} &\mapsto 16 \\ \textit{ig-synchro} &\mapsto 7 \end{aligned}$$

and the synchronization words are—they are given using hexadecimal codes—:

[("F5", "F6", "C0", "C1", "C2", "C3", "C4"),
("C5", "C6", "D0", "D1", "D2", "D3", "D4"),
("D5", "D6", "E0", "E1", "E2", "E3", "E4"),
("E5", "E6", "F0", "F1", "F2", "F3", "F4")]

3 Specifying an application family

3.1 The ways to reuse

By using our framework for developing telemetries, an environment for this comprises the generic telemetry and instantiations for obtaining some telemetries. Each instantiation leads to a specification which is *implemented* using a programming language. This global situation is depicted in Figure 3. Two possibilities for reusing exist.

Reuse of the generic part If the used programming language supports genericity (like Ada), the operators of which we give the behaviour (e.g. the chronological sort) can be implemented once and reused for each new telemetry.

Reuse of implementations If we find out—when we integrating it—that the instantiation of a new telemetry is the same as a previous one, then reuse of implementation of this previous telemetry is allowed.

This second case can be extended when only *entities* are identical, but the implementation process must have respected the specification modularity in order that reuse is possible.

In order to guarantee reuse, it is *crucial* that the consistence are maintained. As a consequence, modifying generic telemetry—for correcting an error, adding a forgotten functionality, or reporting a standard change—should be followed by an up-to-date about all the instantiations. Otherwise, it is desirable to consider the "new" generic telemetry specification as a core of a new tool.

3.2 Lessons

If we try to summarize the presented application, several lessons have been learned. According to what has been done, we can affirm that using a very high formalism for expressing of real problem is mandatory for many reasons·

- the establishment of a real specification in which every thing is explained and where no ambiguity remains;
- an efficient means for developing a software family where the main keyword is reuse;
- a first prototype of the system has been developed;
- writing a formal document which represents *what the order wants* in a very clear manner. This document may be considered as well as a reference document for the following steps of the Software Life Cycle. As main consequence, this document can also be reused along the development process for verification and validation purposes

Moreover we have shown that using a peculiar formalism such as an algebraic specification language is not too difficult if the semantic distance between the application and the chosen formalism is not too high

We have been very surprised from our viewpoint to see that telemetry specialists have accepted to learn the (very strange) formal language and are now able to read our telemetry specifications.

Conclusion

Now writing a formal specification is becoming a feasible task at the industrial level. Some large applications have been described with algebraic specification. As an example, a specification of the File Management System of UNIX is described in [3]. The purpose of that specification was to study the ambiguities, contradictions, incompletenesses, inconsistencies of part of an operating system such as UNIX We will say that kind of formal specifications have been written in order to evaluate all the power of formal specifications on a concrete example.

In our context, the CNES which is more a client than a developer had put the main emphasize about maintenance and reuse of software systems. The formal general specification—describing an application family—it is not too difficult to maintain and in case of any modification at the highest level, it is easy to measure the consequence on the final codes when these codes have been implemented by following the specification modularity. In case of development of a new but similar system an instantiation is straightforward.

Nevertheless a few drawbacks do exist.

The first one is about the kind of languages we can use. These languages are generally based on mathematical notions They are not readable enough as should be dedicated interfaces.

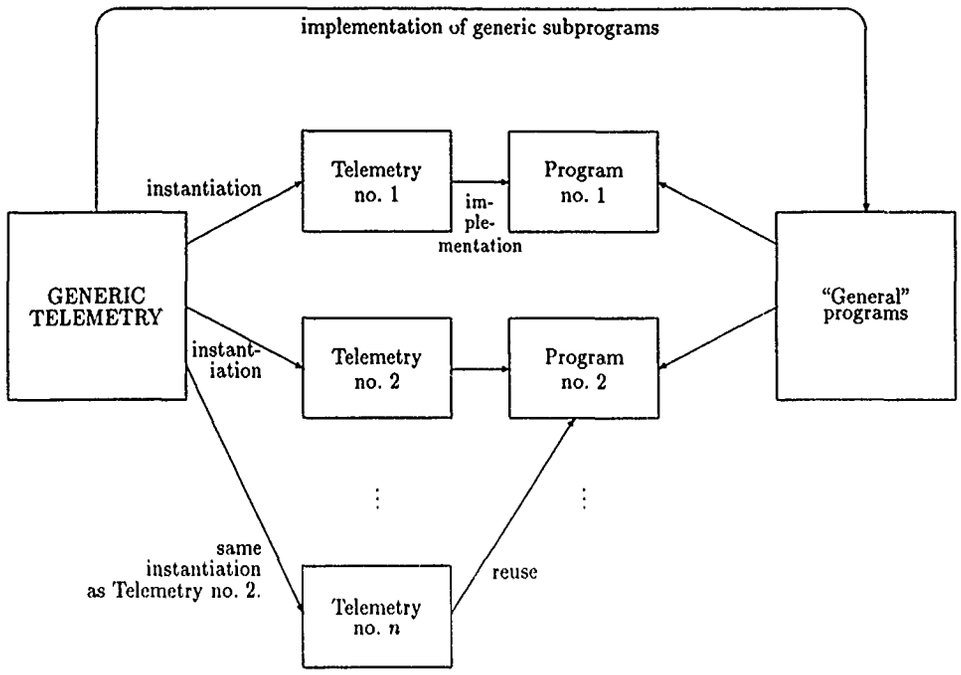


Figure 3: Integrating a new telemetry with reuse.

The second drawback is related to the lack of industrial environment supporting formal languages and formal methods. Indeed, even if formal methods such as VDM [11] do exist and are in practice, they suffer from a lack of support tools.

The third one is about the expressive power of formal languages. As it can be read in this paper, functionalities are easily described. What about the operational properties of a system? By operational properties is meant real time constraint such as *time* and *space* but also other properties such as *friendliness* of the interfaces.

All the drawbacks will be overridden very soon.

The first point is currently being solved by education. The academia has started teaching formal languages and formal methods as well a few years ago. The new generation of computer scientist will be able to tackle the problem of specifying formally.

The second point is fully considered by the industrial world. A few environments have been developed in large European project such as PROSPECTRA [12] and are ready for industrial use.

The third point is the only one for which only partial solutions exist.

Finally, we have shown that algebraic specifications could be successfully used within spatial domain to describe an *application family* in a precise and reusable way. Consequently, we can think we have answered the preoccupations of the CNES.

Annex

Hereafter we briefly expose our notations about genericity. The foundations of this theory can be found in [4]. Since it increases the expression powerful, it has been integrated in most of present algebraic specification languages, e.g ACT ONE [4], OBJ3 [7], PLUSS [2], LPG [1] ..

For the specifications we have written, we have used FP2 (Functional Parallel Programming.), or more exactly a subset described in [8] and in [9, Annex B] (This generic specification language includes also some aspects of parallelism [13].)

In FP2, parameter specifications are expressed by means of *property* modules The types and operators introduced inside this module are given after the module name, e.g.—"/" is a syntactic separator between types and operations—·

$$Total\text{-}Order[t \ / \ rel, eq]$$

where t is any totally ordered set, rel is a total order relation, and eq is an equivalence one (eq is needed

to specify the antisymmetry of *rel*). For any instantiation, the lexical order is used to substitute formal parameters by actual values, e.g.:

$$Total\text{-}Order[Nat\ /\ \leq, =]$$

which points that the type *Nat* of the natural numbers, equipped with the boolean operators "$\leq$" and "$=$" is a totally ordered set.

Three types which appear in the *Satellite-Feature* property (cf. Figure 2) are generic.

- The fixed arrays: they are parameterized by the property *Ftype-and-Nat* which comprises the constituent type and the natural number which represents the dimension. Thus, the expression:

  $$Fixed\text{-}Array.Ftype\text{-}and\text{-}Nat[Bit\ /\ 10]$$

  denotes the fixed arrays whose constituents are bits and dimension is 10 (decimal integers are allowed).

- In the same way, matrices are parameterized by a property which includes the constituent type, the number of rows, and the number of columns, e.g.:

  $$Matrix.Ftype\text{-}and\text{-}2\text{-}Nat[Byte\ /\ 32, 16]$$

  which is a type expression used for specifying the INTERBALL telemetry format.

- "*Seq*" denotes the generic sequences. They are parameterized by a property which includes one type. Such sequence types can be noted in an abridged way, e.g. "*Seq[Bit]*".

(All the rules about the conventions for these type expressions provided by FP2 can be found in [8])

Let us note also that parameter specification can import another parameter specification by means of an *inclusion* mechanism, as exists in object-oriented languages. (We use this feature in Figure 2.)

References

[1] BERT (Didier), DRABIK (Pascal), ECHAHED (Rachid), HUFFLEN (Jean-Michel), DECLERFAYT (Olivier), DEMEUSE (Brigitte), SCHOBBENS (Pierre-Yves), WAUTIER (François): *Reference Manual of the Specification Language LPG. Version 1.8 on SUN Workstations.* LIFIA, RT 59. Grenoble, March 1990.

[2] BIDOIT (Michel): *PLUSS, un langage pour le développement de spécifications algébriques modulaires.* Thèse d'État. Orsay, mars 1989.

[3] DECLERFAYT (Olivier), DEMEUSE (Brigitte), SCHOBBENS (Pierre-Yves), WAUTIER (François): *Adéquation des spécifications formelles aux problèmes de grande envergure.* Software Engineering & Its Applications. Second International Workshop. Toulouse, 4-8 December 1989. Proceedings, Vol. 1, pp. 483-505. EC2.

[4] EHRIG (Hartmut), MAHR (Bernd): *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science, Vol. 6. Springer-Verlag, 1985.

[5] GIROD (Françoise), THOUVENIN (Jean-Pierre): *Une télémesure : qu'est-ce, d'où ça vient, où ça va ?* Communication CNES, janvier 1990.

[6] GOGUEN (Joseph A.), THATCHER (James W.), WAGNER (Eric W.) *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* Current Trends in Programming Methodology. Vol. 4: Data Structuring, chap 5. Prentice-Hall, 1978.

[7] GOGUEN (Joseph A.), WINKLER (Timothy C.): *Introducing OBJ3.* SRI-CSL-88-9 SRI Projects 1243, 2316, and 4415. August 1988.

[8] HUFFLEN (Jean-Michel): *Fonctions et généricité dans un langage de programmation parallèle* Thèse de l'INPG. Grenoble, juillet 1989

[9] HUFFLEN (Jean-Michel) *Réutilisabilité & télémesure. Utilisation d'un outil de spécification algébrique.* Rapport de fin de contrat CNES, septembre 1990

[10] HUFFLEN (Jean-Michel), GIROD (Françoise), THOUVENIN (Jean-Pierre): *Une utilisation industrielle des spécifications algébriques dans le domaine spatial* To appear in Proc. CIL'91 Barcelona, May 1991

[11] JONES (Cliff Bryn): *Systematic Software Development Using VDM.* Second Edition, 1990. Prentice Hall, Series in Computer Science.

[12] KRIEG-BRÜCKNER (Bernd): *Formalisation of Developments. an Algebraic Approach.* In "ESPRIT'87, Achievements and Impact, Part I", pp. 491-502. North-Holland, September 1987.

[13] SCHNOEBELEN (Philippe), JORRAND (Philippe): *Principles of FP2: Term Algebras for Specification of Parallel Machines.* In. "Languages for Parallel Architectures: Design, Semantics, Implementation Models" Wiley, 1989

FORMAL VERIFICATION OF A REDUNDANCY MANAGEMENT ALGORITHM

by
Jonathan Draper
Systems Engineer
CEC Avionics Limited
Technology and Systems Research Laboratory
Airport Works
Rochester
Kent
ME1 2XX
UK

SUMMARY

This paper describes work on mathematical formal verification of a redundancy management algorithm that was carried out in two stages. The first stage used the specification language Z and verified the specifications with hand written rigorous proofs. The second stage used a proof tool to produce formal proofs and specified the system with the language of that proof tool. The system specified was part of a safety critical software section of an avionic system.

The paper includes a section that presents the theoretical concepts of formal methods, concentrating on specification and proof. These ideas are illustrated in the paper with extracts from the formal specifications. Some of the benefits and problems of using mathematical proof for verification are described in the illustration of the redundancy management example.

INTRODUCTION

This paper is divided into two main sections: the first on theoretical concepts and the second describing the work done with the examples. The conclusions of the work are given at the end of the paper.

The first section of the paper starts by defining the terms verification and validation. It continues by describing the main ideas of formal methods: formal languages, formal specifications, formal requirements, formal refinement and proof.

The second section of the paper reports on the two stages of work done on the example - the redundancy management algorithm. The different results of the stages are discussed.

FORMAL TECHNIQUES

Validation and Verification

Validation and verification are processes that demonstrate the correctness of a design. However, as there are many definitions of these terms, the definitions used in this paper are given below.

Validation is the checking of a design against the real world: does the system performance satisfy the customer? An example of validation is the inspection of a high level requirements document. The ideas expressed in the requirements documents are validated, for compliance, against the customer's ideas.

Verification is the checking of one level of design against another level: does the lower level of design satisfy the requirements of the higher level of design? An example of verification is the review of code against a low level design document. The algorithms used in the code are verified against the algorithms required by the low level design.

Validation of a detailed low level design is often performed in two stages: verification against a higher level of design; and then validation of that higher level of design. This is done because validation of the higher level of design is easier as the requirements are not being obscured by implementation detail. For example consider the validation of code, the low level specification, using pseudo code, the high level specification. The code can be verified against the pseudo code by review. Thus the problem of validating the code directly has been simplified to validating the higher level pseudo code.

Thus formal verification is used to show the compliance of a (detailed) low level design with an (abstract) high level document that can itself be validated as containing all the important properties required of the system. This is an idea that will be used later, with the levels of design specified formally and the verification performed with mathematical proofs.

Specification

A formal specification is a description of the design of a system at a given level of detail. It is written in a language with a mathematically formal definition of both the syntax and semantics. Examples of formal languages are VDM or Z but also include subsets of most programming languages. The description is usually of the functional aspects of the system, but may include non-functional details such as temporal requirements. Tools can be used to check that specifications obey the syntactic rules of the specification language, and can also check some of the semantic rules, such as type matches. The levels of design that are specified can range from the highest level safety requirements down to the executable program code.

The initial advantage of formally specifying a level of design is that it forces choices to be made, and recorded, about unclear aspects of the design. The formal specification itself is precise, and can be made abstract. Precision is useful as it removes ambiguity - all the choices that are being left to lower levels of design are made clear. Abstraction is useful in high level specifications, as it allows important properties to stand out.

Where two levels of design are formally specified then formal verification techniques can be used to encure that one is a correct representation of the other. This leads to an approach where successive formal specifications are written, and each is checked against its predecessor. This approach is known as formal refinement or reification. During this approach the formal specifications of the design becomes more detailed and explicit. Thus, the abstract requirements that were clear in the high level specification may be obscured by this detail. Hence, the need to verify the final refined design, using mathematical proof, against the higher level designs.

Proof

Mathematical proof is a formal verification technique. It can be used to show that one mathematical statement follows logically from another. Thus it can be used to verify that a low level formal specification meets a high level formal specification. It can also be used to help validate a specification showing mathematically that a design has desired properties. Proof is used to verify that the specification meets this property; then the mathematical statement of the property can be validated.

A mathematical proof can be presented in either a rigorous or a formal style.

A rigorous proof is an outline of the major points of a proof. It should provide enough detail to enable a formal proof to be constructed. However, the drawback with rigorous proofs is that they are difficult to check, since a mathematician is needed to construct the missing steps and check that the proof is correct.

A formal proof has all the details of the proof at every stage - this includes the proof rules used and the statements they were used on. The steps of a formal proof are exceedingly small (such as the rule in Figure 1) and checking is a simple pattern matching exercise. This can be carried out by non-mathematicians or a simple tool. However, because of the extra detail, more effort is needed initially to write the formal proof.

More advanced tools can help write proofs by:

- Displaying clearly the current stage in the formal proof.
- Applying a rule chosen by the user, calculating the next stage of the proof.
- Storing a library of rules, to allow the user to search for rules that may be useful.
- Let the user define new rules from a combination of old rules and store these in the library.
- Allow the user to write functions that try sets of rules and apply them if they are useful.

A completed proof will have analysed the theorem for every possible combination of inputs and states. This can be contrasted to testing where only a representative, finite set of cases is analysed. However, the theorem may have explicitly excluded certain cases, and the analysis with proof only checks that the theorem is correct in the cases that have not been explicitly excluded.

EXPERIENCE OF FORMAL VERIFICATION

Having described the theoretical ideas behind formal specification and proof, the paper now discusses the formal verification of an example redundancy management algorithm.

This is an example of a top-down proof rule. The rules states how the correctness of the current goal, above the line, can be simplified to the subgoal below the line. Each goal is a list of hypothesis and a conclusion: the goal is true if the truth of the hypothesis entails the truth of the conclusion.

In the rule given below, B and C are general predicates, while Γ stands for all other predicates in the hypothesis list. The rule states that to show that "if the hypotheses in Γ are true then the statement $B \Rightarrow C$ is true" it is sufficient, because of the rule, to show that "if the hypotheses in Γ are true and B is true then C is true". The rule is extremely simple, but a large number of such rules can prove useful statements.

$$\frac{\Gamma \vdash B \Rightarrow C}{\Gamma, B \vdash C}$$

Figure 1: Proof Rule

Overview of System

After an initial study of formal verification on a small subsystem, it was decided to apply the formal verification techniques to a much larger sub-system. The initial work was specified in Z with hand written rigorous proofs. The subsequent work used a proof tool to write formal proofs and specified the system in the language used by that proof tool. This language is based on the functional language ML.

Redundancy management of duplicate resources is an essential part of many safety-critical systems, and this aspect of the system was chosen as an example to study the results of formal verification techniques. Safety critical systems are viewed as a natural application area for formal verification. In particular the algorithms used in safety critical systems must also be considered as safety critical, and it is the algorithm that the formal verification was concentrated on.

With the benefits of the initial study it was decided to use formal verification over a larger area of the system, to specify more algorithms. The main benefits expected were an unambiguous specification of the algorithm and better analysis of the algorithms than testing alone can provide.

Redundancy management, as the name suggests, chooses between redundant elements of a system in the event of failure. The illustrative example chosen is a triplex digital system. There are three lanes, each of which generates commands and qualifies these commands with binary flags. Redundancy management is performed by the comparison of these commands together with exchange of opinions between lanes on the correctness of the other lanes. Ultimately these opinions can cause one of the lanes to stop generating commands. It is the function that generates these opinions that we have specified and validated, and it is clear that such switching of resources is critical to the safe operation of the system.

Overview of Process

Formal verification techniques were used together with informal techniques. Important parts of the algorithm were formally specified, and an informal list of specification assumptions produced. High level requirements were stated initially as broad mathematical theorems. These were modified after analysis to provide a record of the coverage of the formal verification.

Specification

The specification approach employed can be divided into three parts: the framework, the requirements, and the algorithm. The framework and algorithm form the main specification, describing the functional behaviour of the redundancy management subsystem. The requirements are of the higher level behavioural properties that are desired for the system. For example a property may state that once a lane has been isolated it will never transmit any commands. Alternatively, the framework and requirements can be thought of as a high level specification, and the framework and algorithm as a lower level refinement of this specification.

The specification framework describes the interface of the redundancy management subsystem to the rest of the system. This includes listing the variables that are input and output, and a definition of their types. The framework also defines how the inputs and outputs are externally linked in the rest of the system. A simple temporal framework is also defined using functions mapping time to the values of the system states. This framework allows the sequences of the inputs and outputs to be defined and can be used to model some of the effects of the asynchronous aspects of the system.

In describing the structure of the rest of the system, abstractions have been used and assumptions made. Abstractions are used to hide details not relevant to the redundancy management and to simplify the system to make analysis easier. For example the redundancy management does not need to know the exact datatype of the commands it is comparing, only that the commands can be compared, and can use an abstract datatype with this property. Also there are many commands that the redundancy management compares but abstractly only one command need be described and the results of the analysis of the one command generalised to the others.

The assumptions that had been made in the specification were recorded and form part of the specification document. The assumptions covered areas such as: the accuracy of the temporal model; the generalisation of analysis performed on one to the other commands, and the correctness of the simplifying abstraction used on the datatypes. An example of an assumption is given in Figure 2.

The assumption is in two parts: the first half of the sentence describes the assumption; the second half give some justification for the assumption. This will be reviewed when the specification is reviewed.

"It is assumed that the flags A and B can be modelled as a single boolean value, as all processes use the disjunction (A or B) of the flags."

Figure 2: Example Assumption

This is an English paraphrasing of a Z schema that describes part of the redundancy management algorithm. The algorithm is defined on the part of the framework describing the inputs and outputs of a lane. This information is contained in the schema *ΔLaneState*, formally defined elsewhere in the specification. The Δ symbol is the standard Z notation of defining an operation with two copies of the system state: one is used to represent the system before the operation, the other represents the system after the operation, the variables of the state after the operation are shown with a dash after the variable name. The operation is defined by specifying how the final state is related to the before state

It can be seen that the new value of *ownAfc*, that is *ownAfc'*, is defined explicitly in terms of the other variables. (*OTHERLANE* has only two values so the forall statements can be simplified.) Thus the value of *ownAfc'* can be calculated from the inputs. This calculation still uses abstract datatypes and operations (operations that are not directly available in programming languages), such as the relation "does not compare with".

OwnAFCGeneration

ΔLaneState

ownAfc' = *notAvailable* if and only if
(forall *lane* of type *OTHERLANE*•
(*inputs(lane)).dataSent* = *sent* and (*inputs(lane)).afc* = *available* and
threatToSelf(lane) = *threat*) or
(forall *lane* of type *OTHERLANE*•
(*inputs(lane)) dataSent* = *sent* and *(inputs(lane)).afc* = *available* and
(oldCommand, ownCommand) does not compare with
(inputs(lane)).command)

Figure 3: Specification of Design

The list of assumptions was found to be useful as a checklist to be used when the specification was reviewed. As the specification is claimed to accurately model the system apart from the assumptions, validating the specification can concentrate on the closeness of the model. The assumptions can be justified informally, complementing the formal analysis that is validating the system.

The framework is written from informal documents that described the system, and validated against these documents by inspections.

The specification of the algorithm describes the internal operation of each lane, in contrast to the framework which contains the external links of the lanes Together with the framework specification, the algorithmic specification provides a complete description of the aspects of the system that are relevant to the redundancy management subsystem.

The algorithmic specification describes how the outputs defined in the framework can be calculated from the inputs defined in the framework. The algorithm is usually defined explicitly but still uses abstract datatypes, such as lists and sets, and abstract operations. (An example of part of an algorithmic specification is given in Figure 3). The use of abstract datatypes allows the algorithm to be described clearly and concisely, and helps with the reviewing of the specification. The explicit nature of the specification also helps reviewing, and allows the use of simple animations, where the specification is translated into a very high level programming language (e.g ML or Smalltalk) and executed.

The algorithm is formally specified from an outline given in an informal document. One of the benefits of formal specification is that it is easy to see the cases that have been left out of the informal documents.

As with the framework, a list of the assumptions made in formally specifying the algorithm is written down. This list of assumption is very useful as a checklist because it includes the basic assumptions that have been made about the system when designing the algorithm.

The specification is used as a requirements document for the software, the lower level design and the final code can be verified, formally or informally, against the specification

The algorithmic specification itself is validated: partially by formal verification against the requirements specification; and partially by review against the informal descriptions of the algorithms.

This is an English paraphrasing of a Z specification of a requirement. The original English requirement was *"If one or more lanes have been shut down, then no other lane will be deliberately shut down."*

Note that the specification uses some previous definitions: *LifeCycle* contains the state of the system; *NotTransmittingAfter* and *Shutdown* define some of the functions used in the specification. The specification is very precise about when the conditions must hold, and what they must be.

Requirement3

LifeCycle
NotTransmittingAfter
Shutdown

---

forall *frame* of type natural number and *laneNumber* of type *LANEIDENT*•
laneNumber notTransmittingAfter *frame* implies
(forall *otherlane* of type *LANEIDENT* and
subseqFrame of type natural number |
otherlane ≠ *laneNumber* and *subseqFrame* > *frame*•
not (*otherlane* shutdownOn *subseqFrame*))

Figure 4: Specification of Requirement

The requirements specification is a collection of high level functional requirements. Many of the requirements are fundamental to the safe operation of the system. The requirements are defined on the inputs and outputs of the lanes defined in the framework, but at a higher level of abstraction than the algorithm (eg Figure 4). Most of the requirements include a set of conditions; the requirement only applies if these conditions hold.

The requirements are written functionally with two levels of functions. The higher level is almost a paraphrasing of an informal requirement, and uses the lower level functions to link the inputs and outputs. The conditions for the requirement are also clearly specified at this level. Then the lower level functions are defined to give an exact definition This form of definition is used to help validation by separating the overall requirements from the definitions of specific terms.

The requirements were originally written from the informal high level philosophy documents and after discussions with the system designers. The requirement were later modified after analysis with mathematical proofs. The modifications added constraints to remove cases that are not important to the operation of the system and are hard to prove. The final requirements provide a clear record of the cases that have been proved, and, in the constraints, a clear record of the cases which have not been proved. When the requirements are reviewed the exception cases must be covered by informal arguments.

Proof

Mathematical proof was used to verify that the algorithmic and framework specifications met the requirements specification. A theorem (a mathematical statement of an important property) was written for each requirement and proved to follow from the lower level specifications.

For the initial study, which had been written in Z, hand written rigorous proofs were used. The proofs were presented in a formal style using small steps to try and make the proofs easy to check. The final result was a proof that alternated between a series of formal steps and rigorous steps. It was found that the formal steps were very tedious to check by hand, and the rigorous steps were hard to check, thus the confidence in the correctness of the proofs was low.

There were two methods to increase the confidence in the proofs, the proofs could be written with either fully formal steps, or large rigorous steps. Fully formal steps would allow a computer aided tool to check the proof. Large rigorous steps would allow a mathematician to check the proof.

As a computer proof tool was available it was decided to write formal proofs. The tool has a number of advanced features, including tactics which automatically combine simple rules, and the tool has an extensive library of basic rules. The proofs were built from many formal steps although, as many of these steps were chosen automatically by the tool, there is not a record of every step. Thus the record of the proof has a rigorous look and can be reviewed by

mathematicians. But the main confidence in the proof is provided by rerunning the proof on the tool using the proof records.

One of the benefits of using proof as a verification technique was that before writing a proof the author would thoroughly informally analyse the theorem. This would often lead to the theorem being altered into a form that was provable. This often took the form of defining exceptional cases where the theorem does not apply. The proof then provided evidence that the analysis had been performed and checked that it was correct.

The verification of the algorithm against the requirements is completed by informally justifying the cases that have been excluded by the theorem. Either the cases are so unlikely that they can be ignored, or effects that are lost because of the assumptions can be used to show the correct behaviour.

CONCLUSIONS

Overall the role of formal verification in the project can be thought of as similar to that of the role of the reinforcing steel rods in reinforced concrete. The formal verification is very strong in the areas it covers, and the informal analysis gives wide coverage but at a lower level of confidence. Together the combination of formal and informal analysis gives higher confidence in the system.

Specific conclusions from the study of formal verification are:

- Formal specification produces a clear precise description of the system, and highlights any ambiguous parts of the informal description.
- A list of assumptions made during formal specification is useful as a checklist both for the formal specification and the original description.
- Formal proofs should be written using some tool support. Hand written proofs should be as informal as possible while still being rigorous.
- The theorems should be written with a wide coverage originally, then exceptions added during the proof stage. These exceptions must be informally analysed.
- Proofs ensure that the algorithms are analysed thoroughly, and provides a good record of these analysis.

A METHODOLOGY FOR SOFTWARE SPECIFICATION AND DEVELOPMENT BASED ON SIMULATION

by
G. Fernández de la Mora, R. Mínguez, S. Khan, J. R. Villa
SENER
Raimundo Fernández Villaverde 65
Madrid 28003
SPAIN

SUMMARY

This paper discusses the methodology presently used for specification and development of guidance and control software (GCS) refered to as the phased approach. This methodology is shown to present basic shortcomings in relation with the requirements specification phase: long development time, reverse engineering tasks and inadequate handling of errors.

In order to solve these problems, a new methodology, the simulation based approach, is presented. This new methodology is based on the fact that any requirements specification for control software is preceded by a simulation task, that includes the design, code and test of the GCS. As a consequence, the GCS is developed twice, once in the simulation, and then in the flight software

The new methodology proposes to build the GCS only once, and through the use of two basic tools: simulation and rapid prototyping, cuts through the main shortcomings of the phased approach.

1. INTRODUCTION

The aim of this paper is to discuss a new methodology for specification and development of guidance and control software (GCS). This methodology is based on the fact that any GCS is built upon a detailed simulation, which includes a functionally correct version of the GCS.

This methodology was tested in the SBG program [1] and is now been applied to a subset of the EJ-200 DECU [2], in parallel with the phased approach based in the DOD-STD-2167A. This software is actually in the coding and unit testing phase.

The new methodology, referred to as simulation based development, proposes as a core principle to reuse the GCS built for the simulation into the real time target. The key term is reuse. If the GCS exists in two versions functionally identical, one in the simulation and one in the target, and both need to be developed, documented and maintained throughout the life of the equipment, why not to produce a single version ?

Advantages are numerous, including shorter development time - code is developed simultaneously with the control laws -, while disadvantages are minor.

In the following lines we will present more in depth those ideas. Section 2 will discuss the phased approach - based on DOD-STD-2167A - used for the complete DECU software development.

Section 3 will present the simulation based approach.

Section 4 will address the application of the simulation based approach to a subset of the EJ-200 DECU SW.

Section 5 will finally present the conclusions.

2. EJ-200 DECU PHASED APPROACH

2.1 GENERAL

The EJ-200 DECU is a digital electronic box, whose function is to control the EJ-200, the engine to power the European Fighter Aircraf (EFA). The DECU software is in the flight critical category (Level 1 SW according to RTCA/DO-178A).

As a consequence, a set of stringent software standards have been built, in line with DOD-STD-2167A. This specification incorporates a methodology for SW development, that we call the phased approach, based in a sequential series of tasks: specification, design, code and unit testing, and formal testing.

The CASE environment is based on EPOS [3], which includes two basic tools: EPOS-R, used for formulation of the requirements, and EPOS-S, used for the design phase. EPOS-R is a semi-formal specification language, while EPOS-S is a formal design language. EPOS-S allows for automatic partial generation of code. Facilities such as requirements tracing or consistency analysis are included.

Those tools will not be discussed further, since their exact nature is incidental to the new methodoly. But a CASE environment is an important help, since requirements traceability, and ease of documentation generation is a must.

2.2 PHASED APPROACH METHODOLOGY

The top level requirements for the engine control system are included in the document " Engine Control and Monitoring System ". Requirements are expressed in purely functional terms, for example: " Overshoot shall not exceed X% of the demand ". Actual Control Algorithms (CA) are not mentioned. Preparation of this document does not require a prior detailed simulation work. It is based mainly on experience, extrapolation of state of the art techniques and equipments, judgement and conceptual design.

The following document to be produced is the " Electronics System Requirements Document " (FRD). This document not only includes functional requirements, but also all the CA defined in an informal language. It cannot be written without a detailed simulation of the engine, its sensors and actuators, and all the required control loops, including analogue - if any - and digital. It is the last contribution of systems and control engineering to the development, and from it the software engineering phases begin: Software Requirements Specification (SRS), Software Design Document (SDD), etc.

Let us study how the Control part of the FRD is built.

2.3 FRD GENERATION

The process of producing the control part of the FRD is as follows: The engine, its sensors and actuators are modelled to a level of accuracy which is a compromise between fidelity and time - both execution and development -. From this simulation a series of lineal or at least simplified sets of equations are derived, and from those the CA are designed. They are then coded, tested and refined within the simulation. Once a satisfactory solution is found, the CA are translated from the simulation to the FRD.

So, the task of producing the control part of the FRD can be itemised as follows:

- Simulation build up.
- An iterative process of design, code and test of the CA until an adequate solution is found within the simulation.
- A reverse engineering process, through which we transform the GCS in the simulation into requirements in the FRD.

This process implies translating a formal language (the simulation code) into an informal one (FRD requirements). Thus, we obtain the result that the very formal and apparently elegant methodology of the phased approach for software development is in fact based on a reverse engineering process. The document to be obtained in the first phase of the development, the FRD, must be preceded by the design, code and test of the same software we are trying to obtain.

And this result not only applies for development, it is also required for software maintenance. If any control characteristics are to be modified during the life of the engine, those are first tested in the simulation environment, and so the CA have to be coded before the new requirements are incorporated into the FRD.

2.4 PHASED APPROACH APPRAISAL

The phased approach to SW development was a major improvement when it was introduced some fifteen years ago, and has increased considerably software quality. Nonetheless, in the case of control SW, it has several shortcomings. The main ones we have identified are:

- The simulation, which is the basis of the control requirements within the FRD, is usually developed with a set of software standards well below those of the flight code. The result is a significant risk of an error been introduced, either in the simulation of the engine, the accesories, the environment, etc, either in the CA themselves. The complexity of the configuration control in this kind of program increases the chance of a simulation induced error taking place.
- Lenghtly development times. The CA have to be transformed into software at least twice (in the simulation and in the embedded software).
- Some errors which can be introduced early in the development process can only be detected in its very late stages, creating potential hazards. We call those errors open loop errors, due to the following:
  - In the phased approach, part of the development is tied up within a closed loop: the code produced can be tested against its requirements, and as a result any error introduced from the SRS downwards will be detected and consequently corrected before hardware-software integration.
  - Open loop errors are those introduced when producing the FRD or the SRS. They cannot be corrected in the SW tests, whatever the severity of those might be. They can only be found in the system tests, after hardware-software integration. The phased approach does not provide any idea on how to solve the open loop errors, except to look for them in the system tests. This solution is inadequate from a program point of view, since the cost of correcting an error increases as an exponential function of the time it takes to be found.
- Open loop errors not only exist in the phased approach, since for example any error in the modelling of the engine might lead to it under any development methodology. But the phased approach has a tendency to increase them through two effects:
  - The phased approach results in requirements expressed in a formal language in the simulation (Fortran for example), being translated into an informal one in the FRD, and then back again into a different formal language (specification language) in the SRS. Those conversions are a high risk area.
  - The phased approach does not provide any mean for checking simulation CA, FRD and SRS consistency before hardware-software integration.

3. SIMULATION BASED APPROACH

The simulation based approach places some requirements on the development of the simulation itself. We will study those in the first place, and then we will proceed to the flight SW development.

3.1 SIMULATION REQUIREMENTS

The simulation includes at least two different Computer Software Configuration Items (CSCI). The first one, which we will discuss in length thereafter, and includes the GCS, is the " common simulation-embedded SW ". The second one, which we call " other simulation SW ", contains everything else, and is made up by the following Computer Software Components (CSC):

- CSC1: Environment (atmosphere, engine and accessories, including hardware of the electronic box, etc).
- CSC2: Simulation of the embedded SW non included in the CSCI " common simulation-embedded SW ". This CSC is very dependent on the extension of the CSCI " commom simulation-embedded SW ", and in the limit it can be reduced to a simulation of the run-time system of the electronic box. In most cases it will also include hardware dependent features, such as the drivers and the Built-In-Test (BIT). If the CSCI " common simulation-embedded SW " is

reduced as much as possible, it will include everything except the GCS, as for example the input signal conditioning or the supervisory logic.

- CSC3: Analysis tools. Those tools help the developer in its use of the simulation, and includes elements such as data presentation, noise evaluation, transfer functions identification, statistical data analysis, etc.

The requirements for the CSCI " other simulation SW " are as follows:

- It should be considered as a Level 2 software according to RTCA/DO-178A. This requirement comes from the fact that errors in the embedded code due to simulation inadequacies can only be detected in the test bed trials with a real engine. The costs - due mainly to program delays - generated by such late errors can be considerable, and are best avoided by raising the simulation quality.

- CSC1 (environment simulation) shall be developed with the aim of been as complete as possible. This results in two favourable effects: on one hand, the GCS can be tested in a more realistic environment, and the side effects not usually accounted for can be explored. On the other hand, the CSCI " common simulation-embedded SW " can only be made larger if the environment expands itself accordingly. For example, the BIT can only be including in the " common simulation-embedded SW " if the hardware failures are simulated in the CSC1.

Requirements for the CSCI " common simulation-embedded SW " are different. It is to be line to line identical in the simulation and in the embedded SW. It shall be built to the same standards as the flight SW, and its documentation, configuration control and quality evaluation are directly applicable to the flight SW.

In fact, until the point where hardware-software integration begins, development of this CSCI for both the simulation computer and the flight computer is only one activity. From now on, we will consider it as a flight SW task, its use for the simulation been just a by-product. Let us proceed into how this SW is developed.

3.2 FLIGHT SW DEVELOPMENT

The embedded SW is made up of two CSCI. The first one has already been mentioned, it is the " common simulation-embedded SW ", and includes the GCS. The other CSCI is " other embedded SW ", and is made up of all SW modules not contained in the first one. It includes at least the run-time system, and probably the drivers, SW related to HW such as BIT, and in general all SW not developed nor tested with the help of the simulation.

The CSCI " other embedded SW " is to be produced according to the rules of the phased development approach, and in particular DOD-STD-2167A. The new methodology does not introduce in it any modification, except the requirement that each embedded box SW has to have at least two CSCI, one developed within the simulation and the second external to it.

The development of the CSCI " common simulation-embedded SW " implies a new methodology. Two aparently opposite requirements are to be met (see ref [3]):

- SW should be generic enough such that changes typically required by control design (gain modifications, inclusion or elimination of a specific feddback variable, etc) could be easily implemented.

- SW should be adequate for a real time application. This usually implies elimination of unnecessary operations, such as multiplication by zero.

The main difficulty is of course not only to solve those requirements, but to develop control SW according to class 1 standards without a well defined FRD to begin with.

Our answer has been to develop an iterative process, which can be detailed as follows:

- All the SW development process, from FRD to CSCI testing, is redone continuously, with a new SW issue been produced every few weeks.

- Each issue is formaly developed in the sense that changes are first introduced at FRD level, and then implemented down through all the stages of the phased approach until CSCI testing. All basic documents: FRD, SRS, and SDD are kept in EPOS and are traceable.

- SW is divided into two different areas: code itself and what we call " initial parameters " i.e., initialisation values of control and logic variables. Those are kept within an assigned memory area, and can be modified without reissuing the SW itself. In this way the system engineers can modify the control laws without continuously changing the SW. The value of those " initial parameters " allows not only to change the numeric value of gains, limits, etc, but also to feedback or not any specific variable, or to switch from a PI feedback to a PID, etc.

- Every SW issue has an original set of values for " initial parameters " which is subject to configuration control. Changes made to these parameters within each issue are not subject to configuration control, and are only recorded through technical notes. Those might lead, when consolidated, to a change request incorporated into the following FRD issue.

This iterative process is in our opinion within the spirit of DOD-STD-2167A, which in its foreword explicitely states that " The contractor is responsible for selecting software development methods (for example, rapid prototyping) that best support the achievement of contract requirements ".

The iterative activity we have just described is performed by two work teams. The first one is made up by an agregate of different specialists: system and control engineers, fluidodynamicists and safety experts. Their first task is to prepare the preliminary issue of the FRD. From then on, they receive the succesive issues of the SW, test it with different values of the " initial parameters " and prepare the following revisions of the FRD.

The second team is made up of software engineers who prepare, as in the phased approach, the SW itself. Due to the CASE tools used, an important part of this task is done automatically, once the first SW version is implemented.

3.3 SIMULATION APPROACH APPRAISAL

The simulation approach methodology for control SW solves most of the shortcomings we found in the phased approach. Its advantages are as follows:

- Simulation induced errors decrease, as more development effort is put into this tool. This result is important since those errors tend to be expensive to solve, as they appear only in the tests with the real engine.
- Open loop errors generated when translating simulation GCS to the FRD and then to the SRS are eliminated, since the flight SW is tested with the simulation from the initial stages of the program. The only errors that can survive undetected through SW testing are those due to simulation inacuracies and errors found in the CSCI " other embedded SW ".
- Development time decreases, due to a combination of factors:
  - Flight SW prototypes are available very early in the program.
  - Less errors are produced, specially those lengthly to solve.
  - The CSCI " common simulation-embedded SW " is only produced once.

This approach has also several shortcommings. Those we have found are the following:

- As more quality is required from the simulation, more effort has to be put into it.
- The very formal precedure for producing the CSCI " common simulation-embedded SW " even before the CA are well defined might appear as a disproportionate effort.
- The embedded SW produced maintains some characteristics of simulation SW and is not optimized from an execution time point of view.

From our experience, advantages are considerably more important than disadvantages. In fact, some effects which might appear as shortcommings have also favorable aspects. For example, the fact that the embedded SW retains some characteristics of the simulation SW implies, among other things, that it is easier to modify than typical flight SW. This might prove of considerable advantage during the life of the engine.

4. APPLICATION

This methodology has been applied to a small subset of the EJ-200 DECU, the control of the Main Metering Valve (MMV) of the Main Fuel Metering Unit (MFMU). The following steps where followed:

- An initial FRD for the MMV was written.
- The interface definition between the simulation SW and the "common simulation-embedded SW " was defined.
- As the simulation was coded in Fortran, and the " common simulation-embedded SW " was in ADA, a pragma construct was implemented.

The first version of the " common simulation-embedded SW " was produced and running before the initial FRD for the complete DECU, using the EJ-200 SW standards was written. An up to date version was in place, documented and tested during the SRS phase.

Changes were easily introduced into the MMV FRD, such as those required by reliability considerations. The MMV SRS went through further improvements, such as eliminating real number multiplications, or more detailed interfaces definition. An important result was nonetheless negative, as we found that developing through the new methodology a small subset of a CSCI was impractical, since many general purpose procedures (such as table interpolation, for example), not originally intented to be part of the module had to be incorporated in it, as those modules where not yet available.

On the other hand, we found, as expected, that the GCS development time could be considerably reduced.

Another important result was that the simulation task itself, though much more formal than in previous programs, was not slowed down, and it even seemed to be actually faster. One possible interpretation is that the system engineers, been freed from the software task of building the simulation, could concentrate on the control algorithms themselves.

5. CONCLUSIONS

The main results we have obtained are as follows:

- In the phased approach development methodology for control software, the FRD is the result of a reverse engineering process: from simulation to requirements.
- In the phased approach development methodology, any error produced when translating the FRD into the SRS or before, will not be found until the real time simulation test takes place, or even later.
- Embedded SW can be split into two CSCI: one of them, which includes the GCS, can be developed through a new methodology which we have called simulation based approach.
- The simulation based approach uses a common SW in the simulation and in the flight computer.
- The simulation based approach has two basic advantages over the phased approach: shorter development time and avoidance of most open loop errors.

REFERENCES AND NOTES

[1] J. L. Quesada, R. Mínguez and P. Segurola, " Guided Weapon Simulation, the SBGL development experience ", in AGARD Guidance and Control Panel, 50th Symposium on " Computer Aided System Design and Simulation ".

[2] Note: The EJ-200 is the engine to power the European Fighter Aircraft (EFA). The Digital Engine Control Unit (DECU) is an airborne electronic box responsible for the Full Authority Digital Engine Control (FADEC).

[3] EPOS-MANUAL Version 5. GPP.

[4] A. Mattissek, " A unified approach to simulation software and operational software ", in AGARD Guidance and Control Panel, 50th Symposium on " Computer Aided System Design and Simulation ".

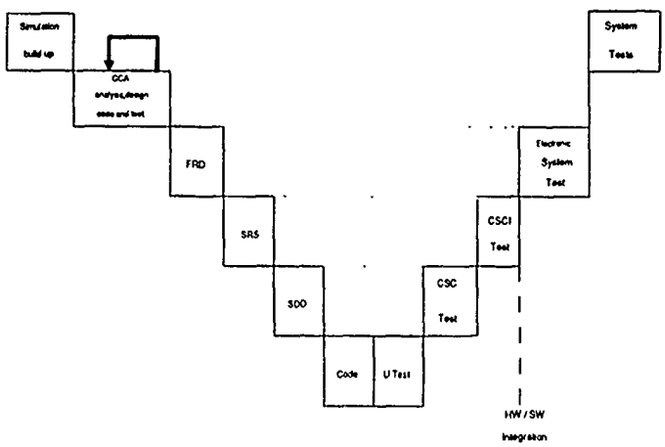


Figure 1: Phased approach methodology in practice

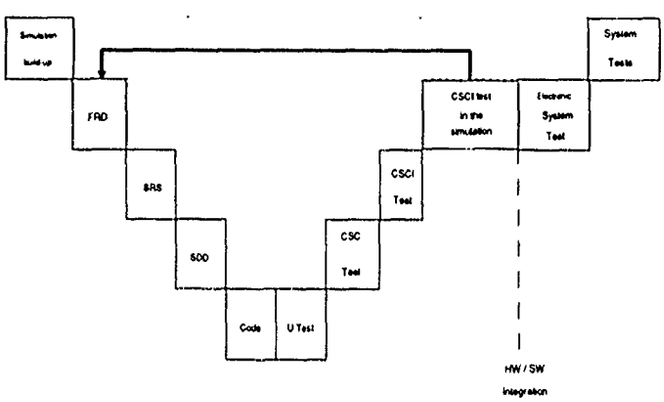


Figure 2: Simulation based approach methodology

NETWORK PROGRAMMING: A DESIGN METHOD AND PROGRAMMING STRATEGY FOR LARGE SOFTWARE SYSTEMS.

by L. Schuberth, J. Kutscher, and W.-J. Grünewald
Forschungsinstitut für Funk und Mathematik
Neuenahrer Str. 20, D-5307 Wachtberg

Summary: Network Programming is a methodology for the evolutionary development and life cycle support of large data processing systems. It utilizes a fully decentralized approach. A given DP task is first realized as an operable network of sequential processes, communicating via typed channels. It serves as a base for logical testing, data flow measurements, and assessment of system behaviour. Runtime requirements and the mapping of processes to processors are taken care of in a separate final step. A remote procedure call illustrates the concept of a channel's operation mode. The Network Programming method is neither confined to a certain programming language nor to a certain kind of machinery. Here we will first give a short introduction to Network Programming and its main features. Then the Network Programmer's Workbench will be shown in some detail and some of its tools will be described. Particular attention will be paid to the Network Monitor. An example will illustrate the use of these software instruments. Finally, we will have a look on a defense oriented simulation.
Keywords: evolutionary software development, communicating sequential processes, typed channels, OSI application layer, Network Monitor.

1. The situation to cope with

Systems for guidance and control are frequently integrated systems with heterogeneous components. Software makes the individual components accessible by describing their interface, and software makes them communicate, thus describing the essence of the system. Software for guidance and control, embedded software, for short, is therefore of considerable complexity. We focus our contribution on large software systems of this kind, which, in addition, have a long period in service.

Large software systems are subject to changes: of the machinery involved, or changes in requirements, or of design goals. Those changes take place usually and should be taken account of from the very beginning of a project. Network Programming (NWP) is a method to construct large software systems upon autonomous communicating processes. NWP helps to cope with all three kinds of changes because it

- supports the implementation of well-defined processes,
- allows processes written in different languages to communicate effectively,
- makes processes communicate effectively, which run on different machines under different operating systems and
- supports changing programs, adding processes to the system, and removing processes from the system at minimal costs in terms of implementation and test times.

The concept of Network Programming has been outlined in [1]. Basic work and an implementation have been reported in [2]. It could easily be used in defense oriented simulations as described in [3]. As a development tool for greater systems was needed, an Ada - oriented workbench had been developed [4]. The use of Network Programmer's Workbench to design, implementation, and test of process networks is described in [5]. In this contribution, the main concept is de-

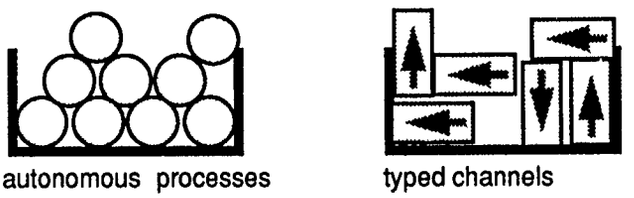


Figure 1: NWP inventory

rived from certain needs when programming large systems.

2.1 Modular Programming: distribution of workload upon dedicated machines

It is a commonplace idea to distribute parts of a complex task to pieces of hardware, which each matches best the details of work to be performed. In reality, i.e. in our context of guidance and control, systems get composed of suitable hardware, and are to be glued together by means of software, later on. It is conceptually no great step to imagine tasks, completely separated one from the other, which communicate by rather simple messages. Each task may be written in its proper programming language, i.e. that optimal medium to describe and effectively attain the individual portion of the general aim.

2.2 Communicating processes by message passing

We regard systems as composed of processes and directed channels (Fig. 1) resulting in a process network, which can be represented as a system communication graph (Fig. 2). Each process executes sequentially and in mutual independence. Currently, it may be described in e.g. Ada, Pascal, C, or Prolog. Other languages may be included by writing an interface in that language and linking it to the application programs.

Each channel connects at least one process to at least another one. It transmits messages of a certain type or of some corresponding structure. The senders and receivers on both ends of each channel command co-operatively the message transfer and must agree upon that message type or structure (e.g. cf. Figure 5).

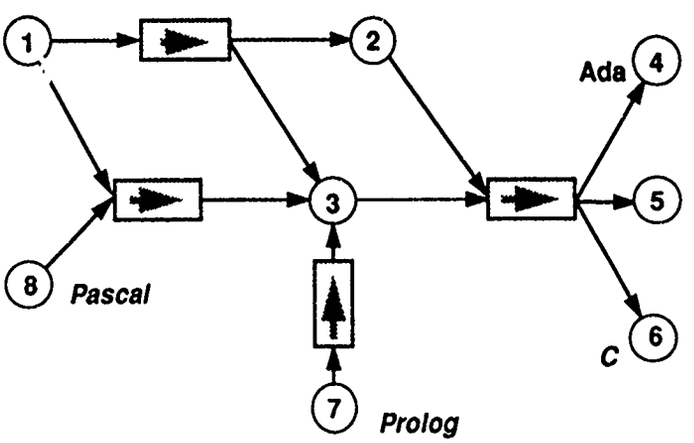


Figure 2: System communication graph

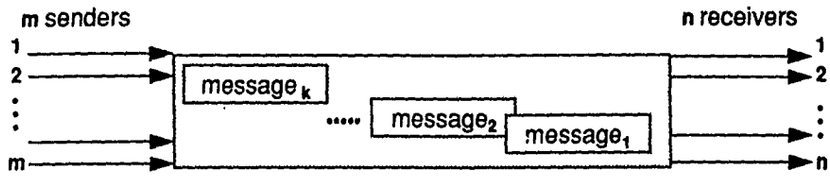


Figure 3: Channel as a typed buffer

The channels are buffers connecting the respective communication partners. Depending on buffer management, there are three connection modes to be distinguished:

- synchronous mode (S mode), buffer is a 0-element queue, the sender waits for receiver;
- buffer-synchronous mode (BS mode), buffer is a k-element queue (cf. Fig. 3), the sender fills up the buffer without hesitation and then waits until at least one element has been removed (read out) by a receiver;
- asynchronous mode (AS mode), the buffer contains one element, which may be overwritten by the sender.

This approach induces the concept of a separate manager: Local Communication System (LCS), to create and maintain the communication ways between the processes, to furnish these channels with appropriate buffer capacity, and to establish the necessary translation between language interfaces. Regarding the last point, we ultimately split the manager. Each process contains a language-dependent part, which transforms language-dependent communication calls to a common communication interface. This part of a process communicates with the LCS and hands messages to and fro.

The communication between different machines takes place as message passing by means of the respective operating systems between the LCSes involved (Fig. 4).

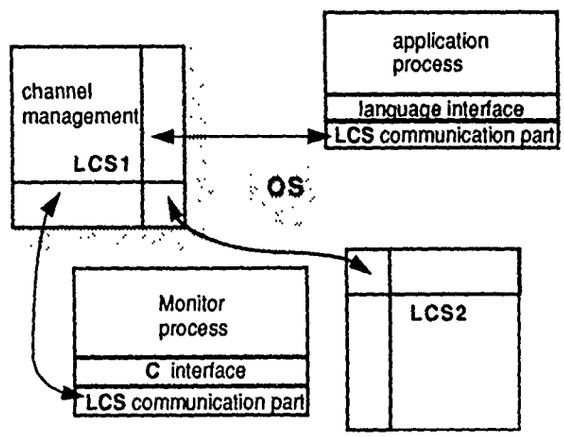


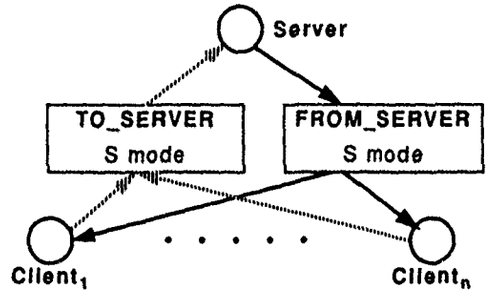
Figure 4: LCS - process connections

2.3 An example: simulating RPC

Nowadays' operating systems comprise remote procedure call (RPC) facilities, and other means to make processes communicate. Fig. 5 illustrates the simulation of a remote procedure call by means of NWP features and their corresponding Ada constructs. The complete listings will be given in the appendix. The lower part of Fig. 5 shows the respective transport commands and their line numbers in these Ada listings.

The subsequent series of Monitor pictures illustrates a process network with one server and three clients under different aspects:

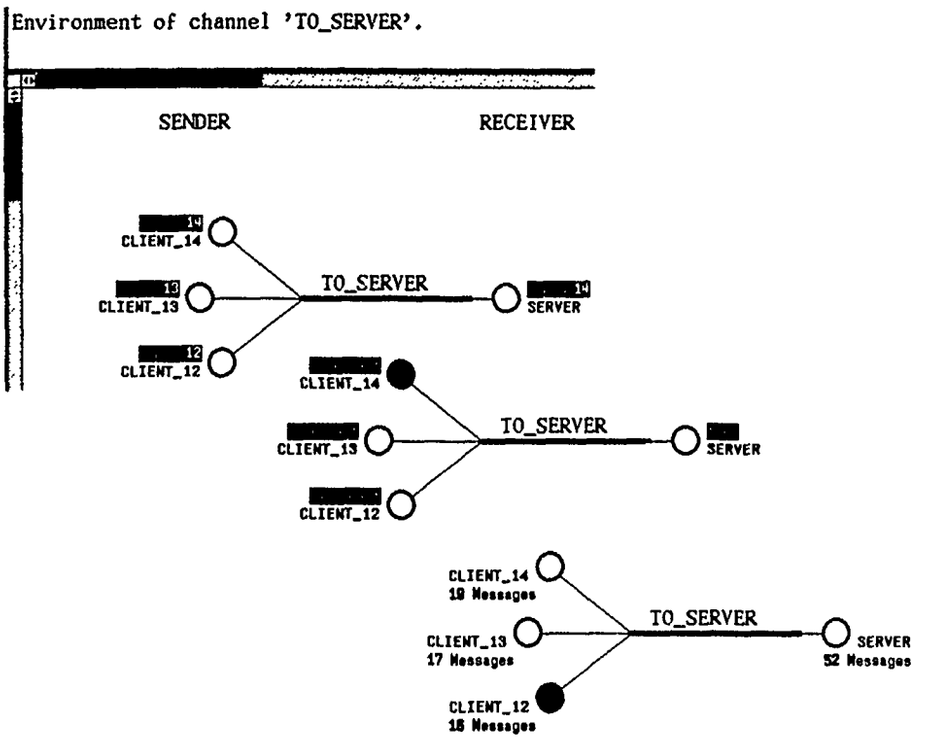
- The network is spread over three machines, the server and one client reside on SUN workstation "rspsun14", while the two other clients run on different SUN workstations, each.
- The clients use the channel in synchronous mode, the access is "give_wait". Client_14 has given a message, which the server has not taken, yet.
- Another state: client_12 has delivered a message, which has not yet been accepted, the flow is 52/22 messages per second.
- The other channel ist FROM_SERVER, it displays again the networks distribution.




```
Client:
    ⋮
40   C.GIVE( TO_SERVER,ORDER );
42   C.TAKE( FROM_SERVER,RESULT,SENDER );
    ⋮
```

```
Server: loop
32      S.TAKE( TO_SERVER,ORDER );
        ⋮
37      S GIVE( FROM_SERVER,RESULT );
      end loop
```

Figure 5: RPC Simulation



52 Messages in 22 seconds

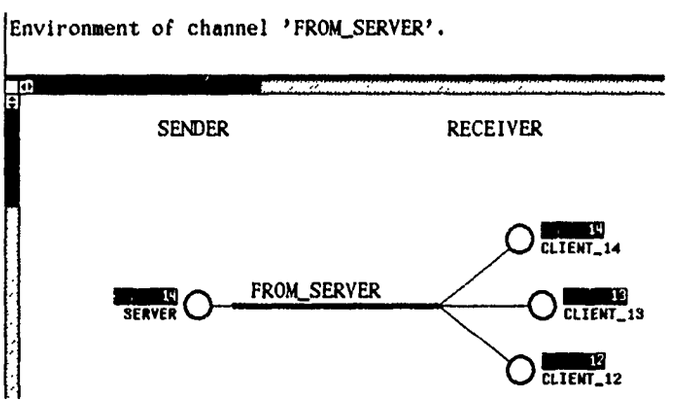


Figure 6: The channels TO_SERVER (above) and FROM_SERVER (below) and their respective environments, access modes, and buffer repletion.

2.4 Systems with different machines under different operating systems

The same idea applies when no common operating system or system family can be assumed: Given a transmission control protocol, the LCSes must adopt the greatest common portion of the protocols shared by the different machines. Our experience comprises:

- processes on different SUN workstations under UNIX
- written in the same programming laguage (Pascal, or C, or Ada, or Prolog)
- written in different programming languages (Pascal, and C, and Ada, and Prolog)

and

- processes on SUN workstations under UNIX and on DEC VAX780 under VMS
- written in VAX-Ada and using the VERDIX Ada Development System,

they all communicate by message exchange. Fig. 7 shows as example a machine configuration in operation at our site.

3. Support environment: the NWP programmer's workbench

Nowadays' operating systems contain virtual file systems, remote procedure call facilities, and other means to make processes communicate. Thus they fulfill OSI specifications on the application layer and encourage the development of distributed data processing applications. To strengthen this gradient of development is one purpose of Network Programming.

Frequently, the partition of work is a major point of concern, be it not to loose human control or understanding, be it to apply specialized hardware, or to avoid waiting situations. If a large system is composed of well fit hardware to fulfill the partitions of task, it

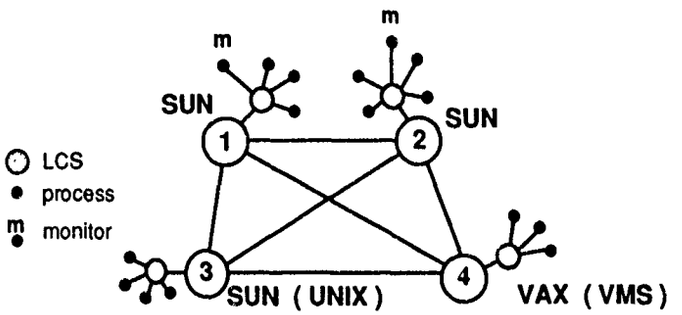


Figure 7: A heterogeneous network example

may occur that in "typical" situations individual components are overloaded and their companions have already finished their part and must wait. Probably the duplication of one dedicated piece of hardware would speed-up the overall performance, or to apply another kind of algorithm, or to build in another switch to discriminate types of situations or of workload. The analogue may arise under conditions near the limit of a system's

area of applicability: in the very beginning, or near system overload, or due to damages. In any case, one will need an embedding system, where the whole object can be run with well defined parameter sets, and can be observed, logged, (statically) reconfigured, and rerun again.

The NWP programmer's workbench was developed to support these doings. The structure of its current Ada oriented realization is outlined in Fig. 8. It currently supports the

- development of process networks

and

- run time analysis of process networks.

The NWP Monitor uses SunViews, the graphics and windows interface of SUN workstations. This notwithstanding, the Monitor can "look" beyond the limits of the SUN subnet of a heterogeneous network. It is language-independent in that all its analysing and evaluating capabilities cover process networks, of which the components are written in every supported language. The constructive means were needed in an Ada centered project, therefore the tools to handle communication channels are Ada directed, and the generator for test environments is Ada directed, as well and for the same reason.

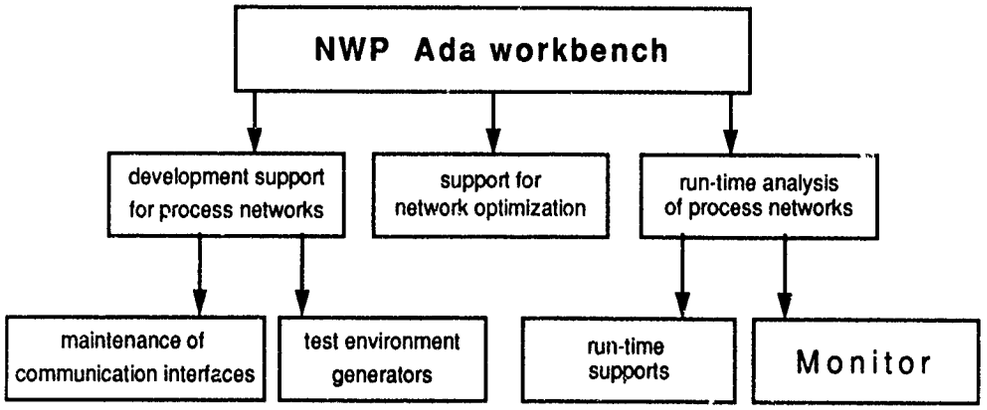


Figure 8: Components of the NWP workbench

For the first kind of work, it offers tools to handle communication channels in Ada and generators for Ada test environments. The second kind of work benefits from NWP workbench's run time support and its Monitor.

The Monitor provides (see Fig. 9):

- static analysis of the network definition,
- recognition of deadlocks and wait situations,
- run time observation of process networks,
- recording and calculation of a communication density,
- process loading support.

It is in terms of the above mentioned "waiting situation" that the enhancement of system performance can be indicated. Thus, in the prototyping phases of developing a large distributed system, the Monitor can supply relative gradients for different configurations of participating processes over a heterogeneous network.

Classical methods of design and analysis are often strictly top down. On the one hand, this causes considerable overhead. On the other hand, it does not lead itself very well to system changes. In fact: each change is likely to modify the original top down structure, giving rise to a respectable increase of system

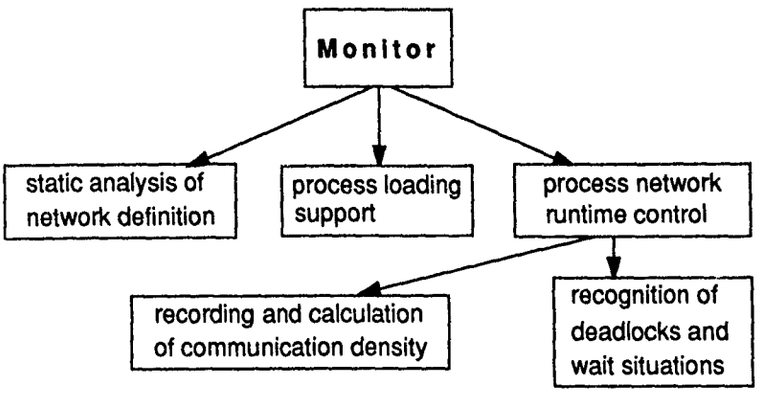


Figure 9: Features of the NWP Monitor

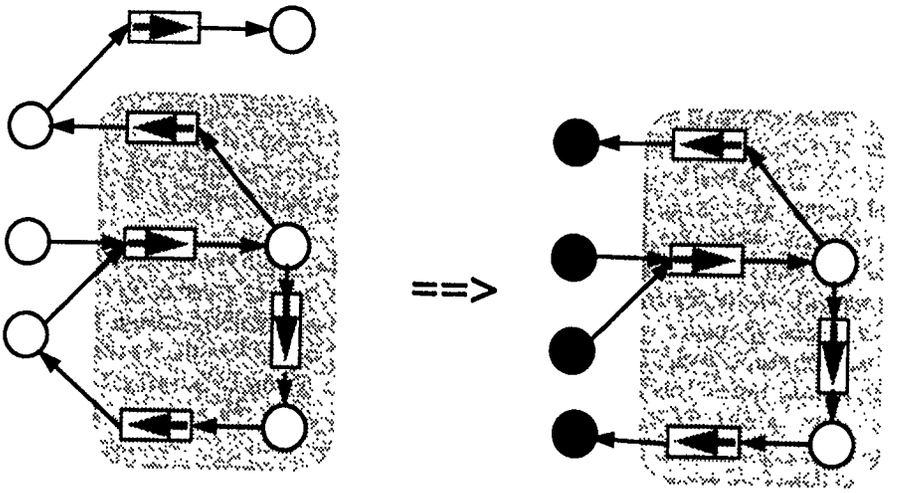
complexity, because maintenance induces degradation of structure [6]. Therefore, the NWP programmer's workbench supports an evolutionary approach and facilitates "rapid prototyping".

To cope with complexity, it is recommended to tailor the component processes of a network to a moderate size and such that bugs can be assumed to be absent. Thus the probability of bugs is moved from the internal structure to the communication of processes, i.e. into the network. Here, because there is no common memory, and because all communication is performed by message passing, debugging is a question of process interfaces, to which the NWP workbench provides automatic support. In addition, since communication is exclusively done by messages, it is unlikely that a bug in one process causes severe effects in another.

The NWP workbench has a toolkit which contains automatic generation of program frames, and a syntax directed editor to declare Ada types and channel masks. Thus we can manipulate processes in order to either change an existing net, or to expand it, or to merge it with another net. The toolkit contains also definition tools for "test cases" or "test scenarios".

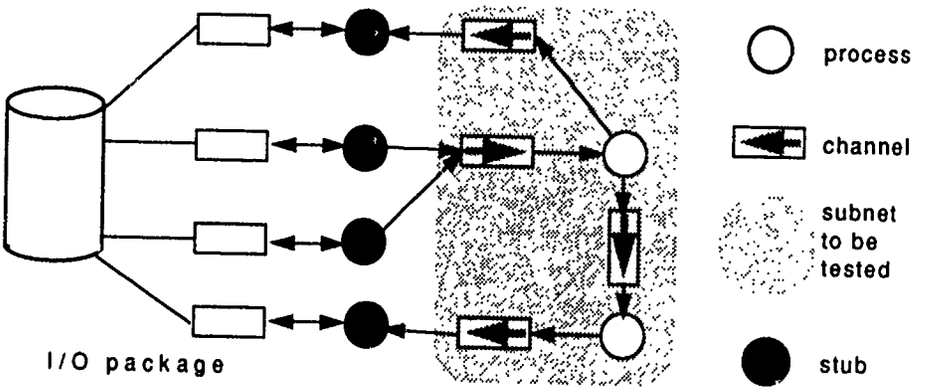
By test case we understand a subnet of a process network, of which the communicative behaviour is to be tested. To do that, data sources and data sinks will be generated automatically, so-called stubs, to shut the open channels after separating the subnet from its surrounding, cf. Fig. 10 a&b.

A test scenario describes the circumstances of performing a test case. The allocation of processes to computers will be defined here, and the co-ordination of stubs, computers, and sets of test data to be run, additionally different operating modes for stubs can be defined here. Syntax directed editors for test data can be generated automatically, which allow to input structured Ada types. Based on the internal representation of Ada data types, such an editor produces an ASCII representation, and thus assures portability of data. A "normalisator" will be generated concurrently as a means to produce an ASCII - file of test data. On the top of this part, a tool to easily produce files of test data is provided. Fig. 10c sketches the test configuration, i.e. the subnet and its interconnection to the test environment, which fully simulates the interface of the original surrounding.



a: network partition

b: subnet embedding



c: test environment

Figure 10: Testing a subnet

4. Modelling naval anti-missile defence

To get an impression of the ability of the NWP workbench, have a look on a Pascal-written subnet of a naval weapons guidance system [7]. This subnet is tasked to detect potential mutual destruction and deflection of own effectors, which are aimed at hostile anti-ship missiles. Different algorithms are to be applied, whether destruction of an anti-missile missile by another one or by gun fire is to be foreseen, and another kind of algorithms, in order to avoid one missile deviating another one, in case the second has a programmed or seeking sensor. Here, the question of parallel computation according to different algorithms, arises.

The Monitor depicts the network and the clustering device can be helpful to judge appropiate grouping of computers and their connection to different sensors. The whole model finds practical application in the decision process to upgrade resp. re-design naval weapons guidance systems.

The first part of the Monitor's screen, Fig. 11, shows the project's name: WW, gives the number of all possible network participants: 11, while already 9 of them have introduced themselves to the LCS, or have been mentioned by some other participant. Ninety channels are foreseen within the process net. Below this window, the whole process network is depicted as a connection graph, with

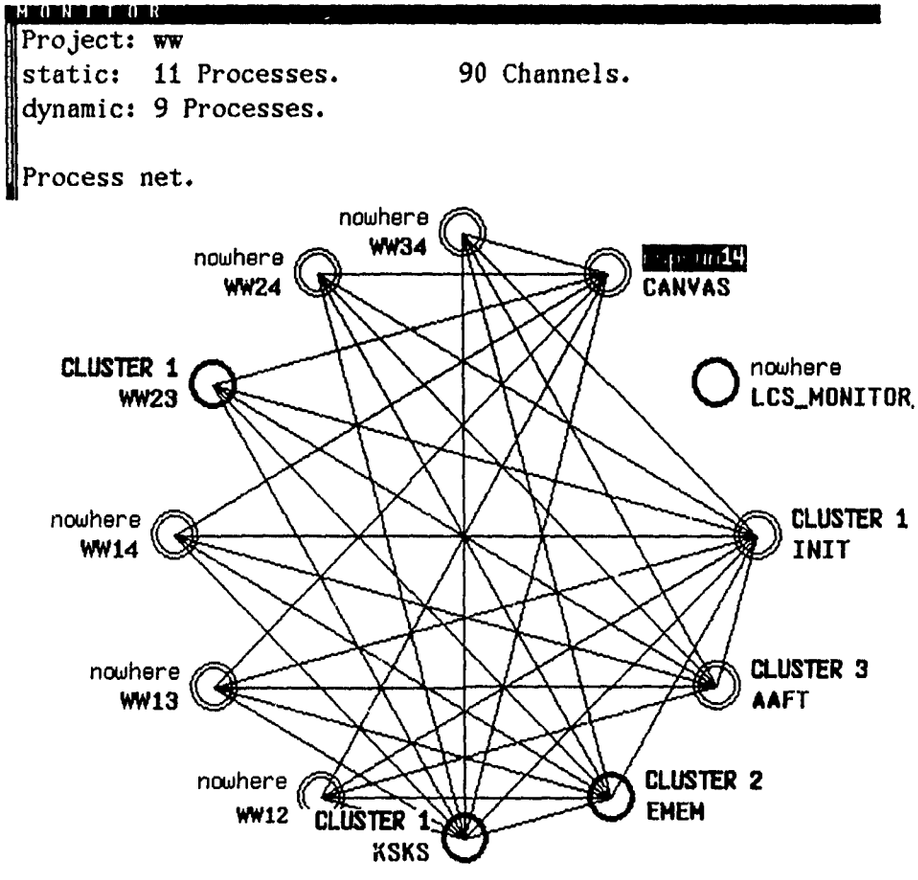


Figure 11: Monitor's state indicating window, and connection graph of project WW

named nodes (processes) and edges to indicate the connections. Solid circles indicate that these processes have already started their computing activity.

The top-left part of Fig. 12 shows a grouping recommendation, after the Monitor has evaluated the communication intensity under the network's initial phase: the process WW23, which has been initialized by INIT, and reporting to the man-machine-interface KSKS, should be run in close connection to INIT and KSKS, on the same machine, if possible. The process EMEM represents the sensors of the weapons system, and AAFT reads an anti-aircraft fire table. The Monitor's cluster analysis proposes to locate them on remote machines, apart from the first process cluster. The lower-right part of that picture shows the table of process affinities (i.e.numbers of channels weighted with communications frequency), which the above recommendation is based upon. This grouping is depicted within Fig. 11, and additionally that network state, when WW23 has been told from KSKS to communicate data to CANVAS, in order to sketch details of a possible influence of one own missile upon the target-seeking sensor of another own missile. Heavy circles mark the already active processes. CANVAS, in it's turn, using SUN View's graphical facilities, must be resident on a SUN workstation: rsp-sun14, in this case, it has not yet begun its activity and its circle is still empty. Because none of the other WW - processes has been started by INIT, their border circles are also empty, and they reside "nowhere". INIT has done it's job and has withdrawn, thus it's border line is empty, too. The Monitor is not a member of the process network, therefore it is also flagged to reside nowhere(with respect to the project WW).

Though we did not perform specific time measurements, we may say that the process communication under LCSes enables fire control to dispose in time over the boat's resources and to avoid own missiles' "fratricide".

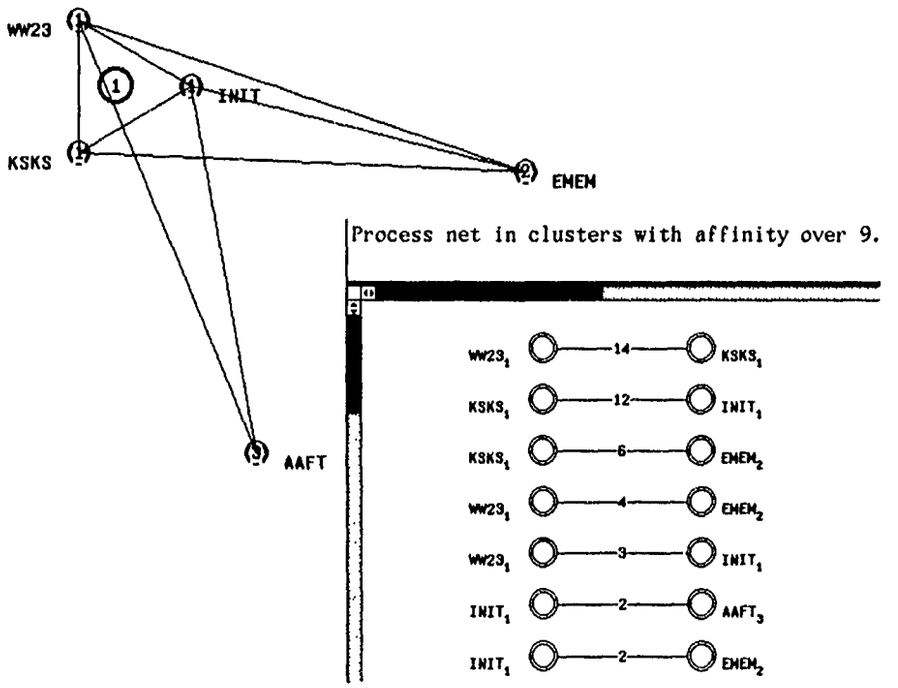


Figure 12: Monitor's grouping recommendation based on process communication history

References:

[1] H. von Issendorff:
Netzwerkprogrammierung - Eine universelle Programmiermethodik
FFM - Bericht Nr.328, Wachtberg, January 1983

[2] W. J. Grünewald:
Netz - Pascal unter BS2000
FFM - Bericht Nr. 352, Wachtberg, July 1985

[3] L. Schuberth: Ein Prozeßnetzwerk zur Vermeidung von Waffen-Wechselwirkungen bei der Verteidigung gegen Seeziel-Flugkörper
FFM - Bericht Nr. 380, Wachtberg, February 1988

[4] W.-J. Grünewald, J. Kutscher, Th. Schell:
Ein Arbeitsplatz zur Programmierung verteilter System in Ada
Wehrtechnisches Ada Symposium, Mannheim, November 1988

[5] J. Kutscher:
Das Entwickeln und Testen von Prozeßnetzen mit dem Netzwerk-Programmierungsarbeitsplatz.
Fachtagung Softwareentwicklung, Informatik Fachberichte 212, Springer Verlag, Juni 1989

[6] M. Hallmann:
Eine transaktionsorientierte operationale Methode zur Anforderungserfassung für das Prototyping (Diss.)
Dortmund, 1988

[7] MTG Marinetechnik GmbH:
FüWES Strukturen --
Operationeller Einsatz Eigenschiffsführung -- Überwasserbekämpfbarkeit
Berechnung der Wechselwirkungen beim Einsatz von Hardkillmaßnahmen,
Hamburg, 1987

Appendix:

Subsequently, the listings of our RPC-simulation example are given. Because this example is coded in Ada, first the language interface for Ada is partly listed, as an example high level language interface.

```
LCS_INTERFACE   1-Mar-1991 19:07:34    VAX Ada V1.5-44
    1 -- LCS_INTERFACE for Ada, 12.12.88                    VMS type
    2 ------------------------------------------------------------
    3 package LCS_INTERFACE is
    4    type CHANNEL_ID  is private;
    5    type CHANNEL_SET is private;
    6    INVALID_MEMBER : constant CHANNEL_ID;
    7    LCS_SYSTEM_ERROR,
    8    LCS_USER_ERROR,
    9    LCS_SIZE_ERROR,
   10    NOT_IMPLEMENTED,
   11    CHANNEL_CLOSED : exception;
   12    type COMMUNICATION_KIND is
   13        ( give_to, give_wait_to, give, give_wait,
   14          write_to, write,
   15          take_from, take, read_from, read );
   ...
   19    procedure LOGON_PROCESS( PROCESS:STRING );
   20    procedure LOGOFF_PROCESS( PROCESS:STRING );
   21    procedure SELECT_CHANNEL( CHOICE     : in CHANNEL_SET;
   22                              WAITING_SET: out CHANNEL_SET;
   23                              COUNT      : out INTEGER     );
   24    procedure SELECT_CHANNEL( CHANNEL    : in  CHANNEL_ID;
   25                              WAITING_SET: out CHANNEL_SET;
   26                              COUNT      : out INTEGER     );
   ...
   32 generic type message_type is private;
   33 package INTERFACE is
   34    procedure GIVE( CHANNEL : in CHANNEL_ID;
   35                    MESSAGE : in MESSAGE_TYPE);
   ...
   51    procedure TAKE( CHANNEL : in CHANNEL_ID;
   52                    MESSAGE : out MESSAGE_TYPE;
   53                    SENDER  : out STRING       );
   ...
   63    procedure CREATE_CHANNEL
   64                         ( MODE         : in COMMUNICATION_KIND;
   65                           SENDER       : in STRING :="";
   66                           RECEIVER     : in STRING :="";
   67                           CHANNEL_NAME : in STRING;
   68                           CAPACITY     : in INTEGER:=1;
   69                           CHANNEL      : out CHANNEL_ID    );
   70    procedure CLOSE_CHANNEL( CHANNEL : in CHANNEL_ID );
   71 end INTERFACE;
   72 ------------------------------------------------------------
   73 -- Functions to handle channel sets
   ...
      90 private
   ...
   96 end LCS_INTERFACE;
```

```
SERVER     1-Mar-1991 18:21:03    VAX Ada V1.5-44

 1 with TEXT_IO,INTEGER_TEXT_IO, LCS_INTERFACE;
 2 use  TEXT_IO,INTEGER_TEXT_IO, LCS_INTERFACE;
 ...
 5 procedure SERVER is
 6 SENDERNAME  : STRING(1..32);
 7 NUMBER      : INTEGER;
 8 TO_SERVER   : CHANNEL_ID;
 9 FROM_SERVER : CHANNEL_ID;
10 package S is new INTERFACE(INTEGER);
11
12 begin
13 -- Start communication with local LCS:
14    LOGON_PROCESS( "SERVER" );
15 -- Define a channel from client processes to this one,
16 --                  channel's name is to be "TO_SERVER",
17 --                  its handle is TO_SERVER, here:
18    S.CREATE_CHANNEL( MODE => take,
19              CHANNEL_NAME => "TO_SERVER",
20                   CHANNEL => TO_SERVER    );
21 -- Define a channel to client processes,
22 --                  of buffer-synchronous mode and
23 --                  channel's name is to be "FROM_SERVER",
24 --                  its handle is FROM_SERVER, here:
25    S.CREATE_CHANNEL( MODE => give_wait,
26              CHANNEL_NAME => "FROM_SERVER",
27                   CHANNEL => FROM_SERVER    );
28    NEW_LINE;
29 -- Listen:
30    loop
31 --    accept a message from a client
32       S.TAKE( TO_SERVER, NUMBER, SENDERNAME );
33 --    Print the client's name and the message:
34       PUT( SENDERNAME ); PUT( " : " ); PUT( NUMBER,5 );
35       NEW_LINE;
36 --    Respond by sending back the message to it's sender:
37       S.GIVE( FROM_SERVER, NUMBER );
38 --    Here an exit should take place if a condition holds
39    end loop;
40 -- Good-bye! to LCS
41    LOGOFF_PROCESS( "SERVER" );
42 end SERVER;
```

```
CLIENT   1-Mar-1991 18:20:29    VAX Ada V1.5-44

    1 with TEXT_IO,INTEGER_TEXT_IO, LCS_INTERFACE;
    2 use  TEXT_IO,INTEGER_TEXT_IO, LCS_INTERFACE;
    ...
    5 procedure CLIENT is
    6 SENDERNAME           : STRING(1..32);
    7 NCLIENT              : INTEGER;
    8 LC, NUMBER, ERGEBNIS : INTEGER := 1;
    9 TO_SERVER            : CHANNEL_ID;
   10 FROM_SERVER          : CHANNEL_ID;
   11 package C is new INTERFACE(INTEGER);
   12
   13 begin
   14 -- Individualization: ask user for a (unique) number
   15    PUT( "CLIENT X: GIVE ME A NUMBER X:" ); NEW_LINE;
   16    GET( NCLIENT );
   17 -- Start communication with local LCS:
   18    LOGON_PROCESS( "CLIENT"&integer'IMAGE(NCLIENT) );
   19 -- Define a channel to server process,
   20 --                  of synchronous mode,
   21 --                  with this process as sender,
   22 --                  channel's name is to be "TO_SERVER",
   23 --                  its handle is TO_SERVER, here:
   24    C.CREATE_CHANNEL( MODE => give_wait,
   25                      SENDER => "CLIENT"&integer'IMAGE(NCLIENT),
   26              CHANNEL_NAME => "TO_SERVER",
   27                   CHANNEL => TO_SERVER            );
   28 -- Define a channel from server process to this one,
   29 --                  with this process as receiver,
   30 --                  channel's name is to be "FROM_SERVER",
   31 --                  its handle is FROM_SERVER:
   32    C.CREATE_CHANNEL( MODE => take,
   33                 RECEIVER => "CLIENT"&integer'IMAGE(NCLIENT),
   34               CHANNEL_NAME => "FROM_SERVER",
   35                    CHANNEL => FROM_SERVER           );
   36 -- Activity: ask user, how many actions consisting in
   37 --           sending the action number should be undertaken?
   38    PUT( "GIVE LOOPCOUNT:" ); NEW_LINE; GET( LC );
   39    for I in 1..LC loop                    -- set action number
   40       C.GIVE( TO_SERVER, I );       -- Send number to server
   41 --    Accept answer from server and print it:
   42       C.TAKE( FROM_SERVER, ERGEBNIS, SENDERNAME );
   43       PUT( ERGEBNIS,5 ); PUT( " from " ); PUT( SENDERNAME );
   44 --    Flag an error, if server didn't answer properly:
   45       if I /= ERGEBNIS then PUT( "ERROR" ); end if;
   46       NEW_LINE;
   47    end loop;
   48 -- Having sent LC action numbers, "Good-bye!" to LCS:
   49    LOGOFF_PROCESS( "CLIENT"&integer'IMAGE(NCLIENT) );
   50    exception  -- In case of an error, print its class and
   51 --                        give farewell to LCS at any case:
   52       when LCS_USER_ERROR => PUT( "USER_ERROR" );
   53          LOGOFF_PROCESS("CLIENT"&integer'IMAGE(NCLIENT));
   54       when OTHERS => PUT( "OTHER EXCEPTION" );
   55          LOGOFF_PROCESS( "CLIENT"&integer'IMAGE(NCLIENT) );
   56 end CLIENT;
```

THE DATA ORIENTED REQUIREMENTS IMPLEMENTATION SCHEME

Christine Thomas
British Aerospace (Dynamics) Ltd
PB 230, Six Hills Way, Stevenage
Herts SG1 2DA, United Kingdom.

1 Abstract

A need has been identified for a generalised approach to the specification, design and development of real time embedded systems. There are many tools that cover different parts of the life cycle. Some of these are integrated to various degrees, but for real time systems it is probably true to say that there is not a set of integrated tools which covers all phases of the life cycle. This paper describes the way that the Data Oriented Requirements Implementation Scheme (DORIS) attempts to remedy this situation. DORIS is an applied research project at British Aerospace (Dynamics) Ltd, United Kingdom.

2 Introduction

DORIS is a set of integrated methods and associated tools for the development of real time, embedded, multiprocessor systems. It covers the whole of the development life cycle from the requirements analysis through to implementation in software and hardware. The main aim of DORIS is to support the development of safe and reliable systems. Particular emphasis has been placed on direct support for the development of multiprocessor systems, reducing timing indeterminancies and supporting the traceability of requirements.

Figure 1 shows the fundamental steps that have to be made when developing a system. DORIS uses two existing methods based on the idea of dataflow: The Controlled Requirements Expression (CORE) [1] is used for the definition phase, and a method known as Modular Approach to Software Construction, Operation and Test (MASCOT) [3] is used for the design phase. These methods have to be extended and adapted so that they can be integrated into the DORIS scheme

Implementation is achieved using a new architecture known as the Data Interaction Architecture (DIA) [6]. The DIA is based around shared memory and offers fully controlled asynchronous communication (in addition to the more conventional synchronous communication). It uses two specially designed chips to support multiprocessor applications.

Many existing proprietary tool sets and methods tie the user into a specific language, host or target. British Aerospace has such a large variety of projects that standardization on any of these is impractical. One of the main aims of the DORIS approach is that it will be language, host and processor independent. This will lead to a standardization across products, which in turn will lead to improved productivity.

3 Requirements Analysis

CORE is both a method and a tool designed specifically for the requirements phase of the development life cycle. The method has been developed by British Aerospace (Military Aircraft) Ltd and System Designers Ltd in the United Kingdom. CORE establishes the actual problem to

be solved, and reduces ambiguities and inconsistencies in the customer's requirements. It also highlights the effects produced by changing the system specifications and formalises these system specifications so that they are understood and agreed by all involved in the project. [2]

CORE consists of a set of defined steps for the development of systems requirements models. It encourages structure and therefore modularity, and identifies the data flow between the elements.

The main concept of CORE is that of 'viewpoints'. These describe the nature and content of the problem as seen from particular points of view. Each viewpoint looks at the problem in terms of the information acquired, the processing of this information and the generation of output results.

CORE avoids using complex mathematical notation, and yet still achieves the necessary degree of formality required for requirements analysis. This makes CORE more acceptable to the engineering community.

Once the behaviour of the system is described, this information can be used as the requirements document for the design teams.

4 Design

Figure 2 shows the structure of the design tool set. Three languages have been provided for the design phase of DORIS. These are:

- **DORIS Design Language (DDL)**
  Specifies the software design of the system (based on MASCOT).

- **Hardware Description Language (HDL)**
  Describes the configuration of the processors, the memories and the interfaces between the memories.

- **Mapping Description Language (MDL)**
  Maps the designer's MASCOT activities into specific processors.

The DDL is an adapted form of MASCOT [3, 4, 5]. MASCOT was developed at the Royal Signals and Radar Establishment in the United Kingdom and has been adopted by the United Kingdom Ministry of Defence as a standard approach for the development of embedded systems. MASCOT is a network approach to software design suitable for multi-processor systems. It gives independence from specific processors and provides a framework for specifying interconnections and interfaces between different software (and indeed hardware) components in the system. It has both a graphical and a textual form, each of which can be derived from the other.

In MASCOT, activities are sequential processes, concerned primarily with performing a single function. Each activity is conceptually independent, i.e. it runs concurrently with all other activities. Activities communicate with each other through shared data areas, known as Intercommunication Data Areas (IDAs).

Certain extensions have been made to MASCOT so that communication between the MASCOT elements in independent of both the hardware configuration and the type of the data being communicated. This has been achieved by adding the idea of a **Route** to MASCOT. A Route is a specialised form of an Intercommunication Data Area, with one input and one output. It enables information to be passed from one activity to another, unchanged. It can be used to provide either asynchronous or synchronous communication between the two processes.

To allow this to be implemented, the concept of type-independence has been added to MASCOT. This allows the data type to be specified at instantiation.

In MASCOT, a template a standard pattern for the design of a component in the system. When the designer wants to use a component of that design, he creates an instance of the template. As part of the DORIS system, it is intended that standard templates will be provided to help communication between activities in the system. These templates will provide four methods of communication from a sending process (the writer) to a receiving process (the reader). These are:

- Fully Asynchronous
  This method of communication allows the reader and the writer to operate independently of each other.

- Conditionally Asynchronous
  This form of communication is a limited form of the fully asynchronous mechanism. It is guaranteed to work satisfactorily if the interval between successive writes is always greater than the duration of any read.

- Loosely Synchronous
  This does not require that the two processes are at the same place at the same time to exchange information. It does however, exercise some constraint over the relative operation of the two processes by limiting the extent to which the production of information can get ahead of its consumption. In effect, it is a bounded buffer.

- Fully Synchronous
  This is a rendezvous, which locks together the operation of the two processes at or during the exchange of information. There must be a point in time at which the two processes meet. Data can then be passed between the two and they can then carry on independently.

In addition, three distribution possibilities will be provided for a Route:

- Private
  The activities using the route are both in the same processor

- Shared
  The activities using the Route are in different processors, connected by shared memory.

- Remote
  The activities using the Route are in different processors, not connected by shared memory.

The software designer will be able to define a Route between two activities, regardless of their mapping into hardware. At build time, the builder will determine the relative distribution of the activities, and will substitute a Route with the necessary distribution. This allows the software to be designed without needing to know how the activities are mapped into the processors. One benefit of this is that the software can be tested in the host, using a private distribution, and then the activities can be mapped into different processors, without requiring a corresponding software design change.

The design tools provided with DORIS will allow the software design and the hardware configuration to be easily changed. It is intended that the basic toolset will contain:

- Graphical Design Tools
- Timing Analysis Tools (SPIRITS)
- Textual Analysis and Checking Tools (DAN, HAN, MAN)
- Build/link tools (Builder)
- Loader/Monitor (DEMON)
- Run-time development tools

It is intended that the DDL, HDL and MDL will have graphical front ends to simplify the network design. There is also a textual form of all three languages. One possible front end to DDL is MADGE (MASCOT Design GEnerator). This has been developed by British Aerospace (Dynamics) Ltd and supports a

graphical form of MASCOT.

The problems of timing analysis are being addressed by a Department of Trade and Industry sponsored initiative called SPIRITS (Supporting Predictable Implementation of Requirements In Timing and Safety). This is investigating the need to develop hard real time systems whose timing and safety properties are known and can be shown to satisfy the existing requirements.

Figure 2 shows the relationship of the textual analysis and checking tools. The DDL Analyser (DAN) checks the DDL text (either generated by the user or by MADGE or similar tool). If the syntax is correct, it places the network connectivity information contained in the text (how the components in the system are connected) into the template database, and creates source files in the selected target language for syntax checking using commercially available compilers. This use of propriety compilers aids the language independence of DORIS.

The Mapping Description Language Analyser (MAN) and Hardware Description Language Analyser (HAN) both store the information contained in the MDL and the HDL into appropriate databases for use by the builder.

The Builder checks that all the activities specified in the DDL are mapped to processors specified in the HDL. It also checks that for all IDAs, an IDA template suitable for the appropriate mapping is available in the database. It generates instances of the templates with the correct connection between activities and IDAs. For each processor in the system, the builder generates a list of activities resident in the processor, and presents the instances of activities and IDAs to the compiler and linker.

The loader/monitor (DEMON) provides facilities to allow multiprocessor loading and host/multitarget communications.

DORIS run time development tools will be provided that provide:

- Detection and reporting of run time errors
- Breakpoint handling
- Memory examination
- User Defined Debug Messages (MASCOT record primitive)
- MASCOT primitive monitoring
- MASCOT execution control
- Timing data collection

5 Implementation - The Data Interaction Architecture

The aim of the Data Interaction Architecture (DIA) is to provide a mechanism to support multi-tasking and multi-processing systems. It uses simple hardware elements, giving predictable behaviour for high integrity systems. The DIA provides direct hardware support for MASCOT designs.

Figure 3 shows the basic configuration of an element in the DIA. Ideally the Central Processing Unit (CPU) is a relatively simple form of Reduced Instruction Set Computer (RISC) in which no use is made of features that introduce non deterministic timing effects including interrupts and caching. More complex computers can be used, but this will make it more difficult to analyse run time properties. [6]

The CPU has a private bus that allows it to be connected to:

- private memory (containing activities and private IDAs)
- Asynchronous devices (polled peripheral devices)
- Synchronous devices (peripheral device generating a stimulus)
- Asynchrorous Dual Port Memory (ADPM, containing Intercommunication Data Areas shared with activities in an adjacent processor)
- Two sorts of specially developed VLSI device, the Kernel Integrated Circuit (KERIC) and

the Communication Integrated Circuit (COMIC).

The Kernel Integrated Circuit supports low level scheduling, providing the multi-tasking facilities needed when many activities are mapped onto a single processor, as well as handling external stimuli (from timers etc.) without using interrupts. It selects which activity is to be scheduled by using built in priority and polling rules. It supports cooperative scheduling in preference to pre-emptive (or interrupt driven) scheduling. Use of the Kernel Integrated Circuit avoids the timing overheads normally found in software based executives.

A processor communicates with an adjacent processor via shared memory. The aim of this is to remove the need for buses, thus eliminating the risk of a "single point of failure". It is intended that the shared memory used in DORIS should be Asynchronous Dual Port Memory (ADPM), although other devices can be used. The Communication Integrated Circuit is used to control the access to the shared memory, allowing activities in adjacent processors to pass data from one to another. An Asynchronous Dual Port Memory without arbitration has currently been selected, which avoids any timing interference. Four different forms of communication are supported: fully asynchronous; conditionally asynchronous; loosely synchronous; fully synchronous.

A facility is provided that allows the COMIC to signal to the KERIC in the destination processor that data has arrived for it, and that the appropriate activity can be rescheduled.

6 Conclusion

DORIS aims to provide support for the development of safe and reliable systems. It does this by:

- Reducing timing indeterminancies
  Co-operative scheduling will be used instead of interrupts.
  No inter-processor buses will be used.
  No caching will be used.
  Asynchronous Communication will be used.

- Multiprocessor Support

  Design, Mapping and Hardware Description languages will be provided.
  The substitution of templates in the builder allows the software design to be mapped as appropriate to the hardware available.

- Traceability of Requirements

  The use of CORE and MASCOT will provide the means of tracing the requirements through to implementation.

7 References

1. Mullery G.P. CORE - A method for Controlled Requirement Specification, Proceedings of the Fourth International Conference on Software Engineering 1979, pp 126 - 135.

2. Cooling J.E. Software Design for Real-time systems, 1991, Chapman and Hall. pp 331 - 333.

3. The Official Handbook of MASCOT (Version 3.1), 1987, Defence Research Information Centre, Glasgow.

4. Simpson H.R. The MASCOT Method. Software Engineering Journal, 1986, 1, (3), pp 103-120.

5. Simpson H.R. MASCOT Real Time Networks in Distributed System Design. IEE Colloquium on MASCOT and Related Issues, December 1990.

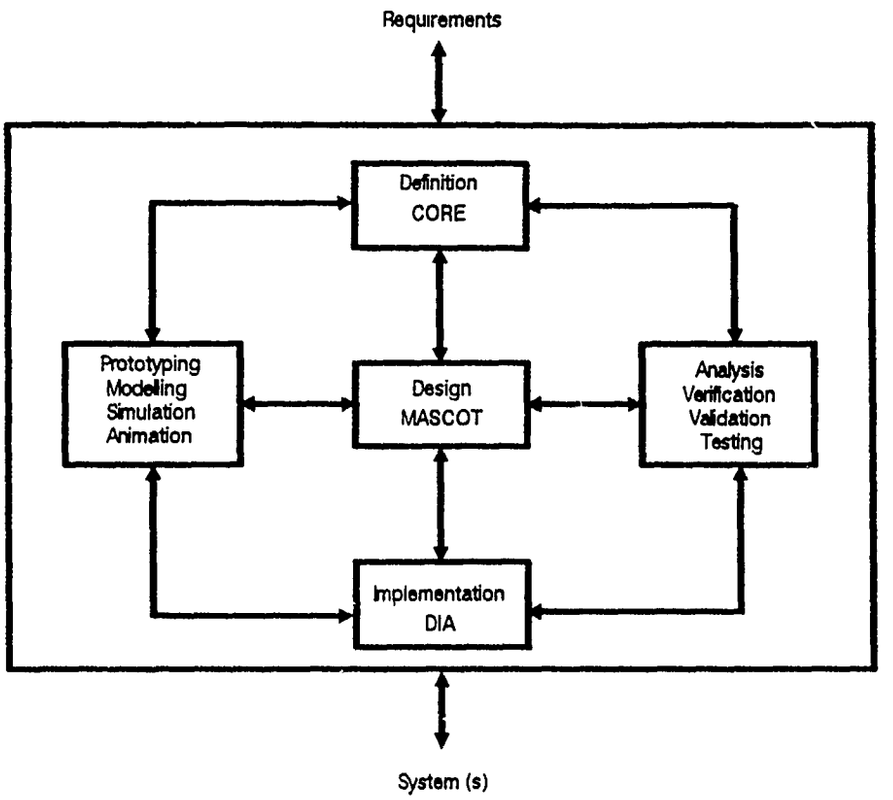
6. Simpson H.R. A Data Interaction Architecture (DIA) for Real Time Embedded Multi Processor Systems. RAe Conference on Computing Techniques in Guided Flight, Boscombe Down, 1990.

8 Acknowledgements

The author would like to acknowledge the help of her colleagues at British Aerospace (Dynamics) Ltd with the preparation of this paper.

The author would also like to thank the Ministry of Defence in the UK for its funding of some of the tools and methods mentioned in this paper.

DORIS DEVELOPMENT PROCESS



CORE COntrolled Requirements Expression

MASCOT Modular Approach to Software Construction, Operation and Test

DIA Data Interaction Architecture

Figure 1

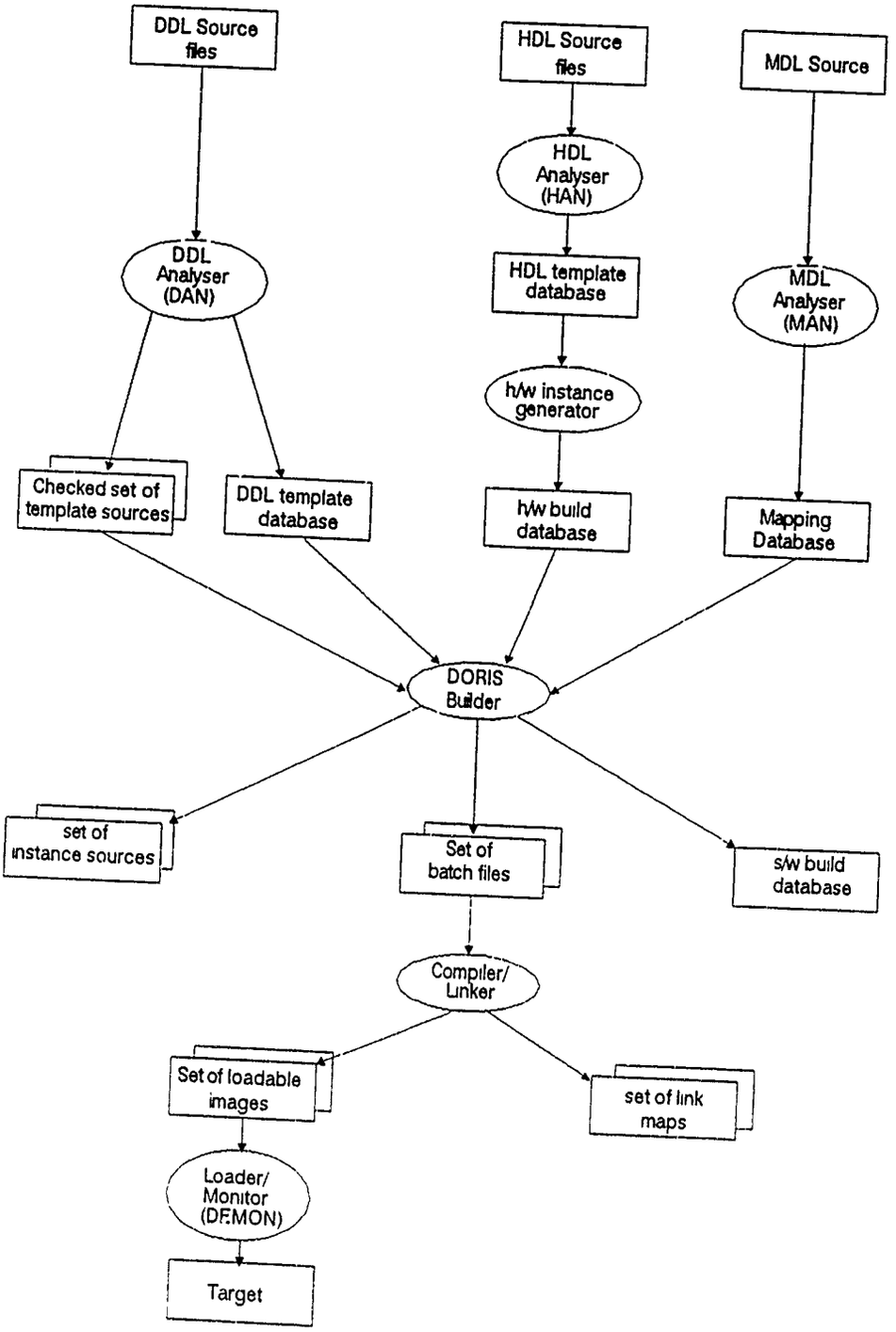
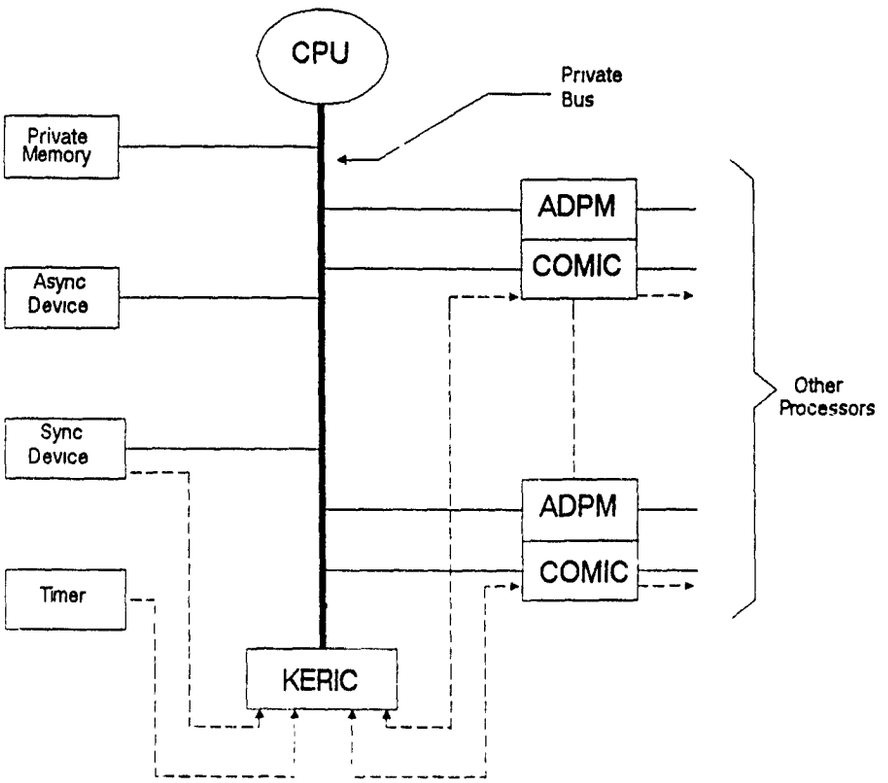

DDL Source files
HDL Source files
MDL Source
HDL Analyser (HAN)
DDL Analyser (DAN)
HDL template database
MDL Analyser (MAN)
h/w instance generator
Checked set of template sources
DDL template database
h/w build database
Mapping Database
DORIS Builder
set of instance sources
Set of batch files
s/w build database
Compiler/ Linker
Set of loadable images
set of link maps
Loader/ Monitor (DEMON)
Target

Figure 2



Connections to Private Bus:

Private Memory: Contains activities and private idas.

Kernel Integrated Circuit: Supports activity scheduling and cooperative external stimuli.

Asynchronous Dual Port Memory: Idas shared with activities in an adjacent processor.

Communication Integrated Circuit: Control and stimulus logic for shared idas.

Async Device: Polled perihperal device.

Sync Device: Peripheral device generating stimulus.

Timer: Periodic stimulus generation.

Figure 3

PROCESS/OBJECT-ORIENTED ADA SOFTWARE DESIGN FOR AN EXPERIMENTAL HELICOPTER

by
K. Grambow
ESG Elektronik-System-GmbH
Postfach 80 05 69
8000 München 80
Germany

Summary

This paper discusses a software design method for real-time applications written in Ada. It proves that even time critical systems can be implemented in pure Ada.

The design method is based on the Ada tasking model in conjunction with object-oriented design (OOD) principles. Special purpose graphs, derived from Yourdon/De Marco data flow diagrams (DFD's), illustrate the method, while Ada program design language (PDL), as a counterpart to the graphs, serves as a basis for the software implementation.

No global cyclical executive is used to schedule the concurrent threads of execution. Instead, a rendezvous-based interaction of Ada tasks provides the scheduling. This is automatically generated from an Ada compiler.

This software design technique is illustrated by the development of the operational flight software for an experimental helicopter.

1. Introduction

In the past, real-time systems were implemented using a dedicated operating system on the target computer or using a higher level programming language with real-time extensions. Since the introduction of Ada, a widely accepted programming language is available which incorporates real-time features such as tasking or interrupt handling in the language itself.

Moreover, complex real-time software development demands a language with design features, modularity concepts and precautions for team-effort implementation. Therefore, Ada was equipped with such features as packages, generics and separate compilation.

These two aspects prove Ada to be a very good tool, both for the design and for the implementation of complex real-time systems.

In this paper a comprehensive design methodology, which strongly utilize Ada's design and real-time features, is presented. The methodology focuses on large real-time systems, giving a practical, step-by-step design approach, which is documented in several graphical illustrations and can be canonically transformed to Ada program design language.

As an example, the design methodology is applied to an experimental helicopter project which is currently under development at ESG.

In the following chapter, we briefly describe the example project. Chapter 3 explains the methodology and applies it to the project. Chapter 4 continues the description of the design, concentrating on refinement steps under utilization of OOD techniques. Chapter 5 is dedicated to real-time scheduling and software performance issues, discussing the question of whether the scheduling based on the Ada tasking model is applicable in time critical systems. The final chapter summarizes the experiences with the design method.

2. A typical real-time project

At ESG, a project is currently under development which equips a helicopter with experimental avionic instrumentation. The helicopter will facilitate the analyses of advanced equipment components and of the man-machine interface aspects of a modern cockpit in the course of flight trials. Modern computer-controlled displays and sensors are at the pilots disposal to gain flight experience in a realistic environment. The results of these flight experiments are a valuable input for and can prove the feasibility of later helicopter products like the German/French "Tiger" anti-tank helicopter.

Due to the experimental character of the project, the system and software design should be flexible and easy to alter or extend. Clearness, ease of change and reusability are important demands on the software development effort. All software for the avionic system is implemented in Ada.

Figure 1 shows the system architecture of this example project:

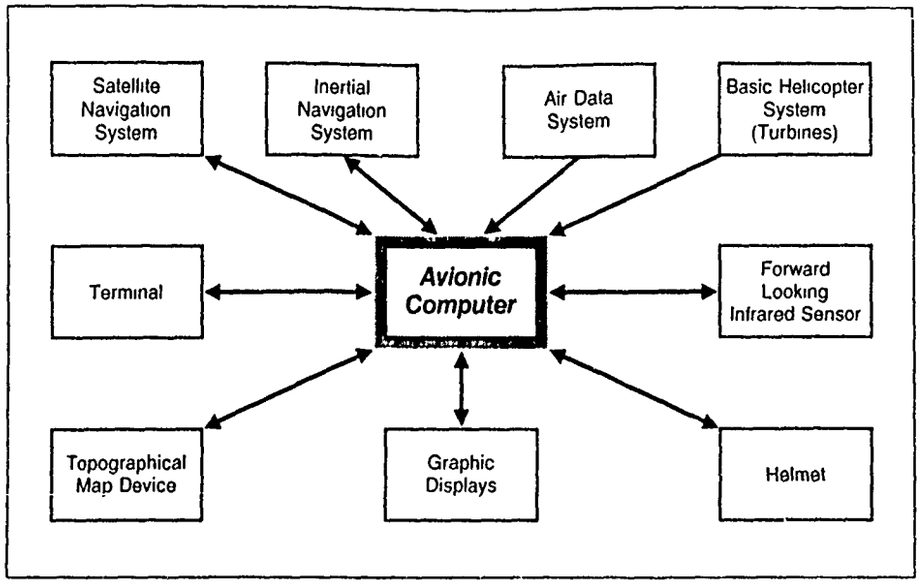


Figure 1: System architecture of the experimental helicopter

The avionic computer is a multiprocessor system, based on Motorola 68030 boards, with an integrated graphic symbol generator. All display and control units, and the sensors, are electrically connected to special input/output boards of the avionic computer.

The pilot controls the avionic instrumentation with a menu guided terminal and with the help of a topographical map device.

All important flight information (flight routes, danger areas, helicopter attitude and velocities) is displayed and continually updated on special multi-color graphic devices.

In addition to the usual inertial navigation system, a very precise satellite controlled navigation system (GPS) is utilized.

Last but not least, the pilot wears a helmet mounted sight/display device: digital information is mixed with the video image of the forward looking infrared camera and can be projected on the screen of the helmet. The camera is controlled through movements of the pilot's head.

3. The Ada software design method

The software design for the helicopter project is derived from a methodology developed by K. Nielsen and K. Shumate at Hughes Aircraft Company (see [1]). It is based on the Ada tasking model in conjunction with object oriented design (OOD) features. Graphics illustrate the global design steps and can be canonically transfered into Ada program design language (PDL). The global design is process oriented, leading to the identification of all concurrent processes and their interactions. In Ada, these are described as tasks and rendezvous. In a refinement step of the design, Ada PDL is further developed to outline the single threads of execution of each task using OOD methods.

In the following, the methodology will be explained step-by-step and illustrated through examples from the experimental helicopter project.

At first, one tries to identify the main software functions and their real-time executions, i.e. whether they perform event-driven or execute periodically. A textual description of the requirements for the project will normally be the basis for this step. As discussed in more detail in chapter 5, no global cyclical scheduler will be used in this kind of design, but all the real-time issues will be handled with the Ada tasking model. Therefore, it is important to fit the main software functions in concurrent processes which will be described as Ada tasks. The real-time execution of these tasks is provided implicitly by the Ada compiler, their interaction is controlled by Ada rendezvous.

But how does one find these tasks? They should be combinations of cohesive software functions. Some will execute periodically, others event-driven. In the latter case, they may be triggered by interrupt. The design methodology defines a comprehensive procedure for the identification of these tasks:

The so-called "edges-in approach" assigns a concurrent process to each of the interface handlers which connect the external hardware devices with the main avionic application. Here, the main part is not yet specified, only its data flows to the outside world.

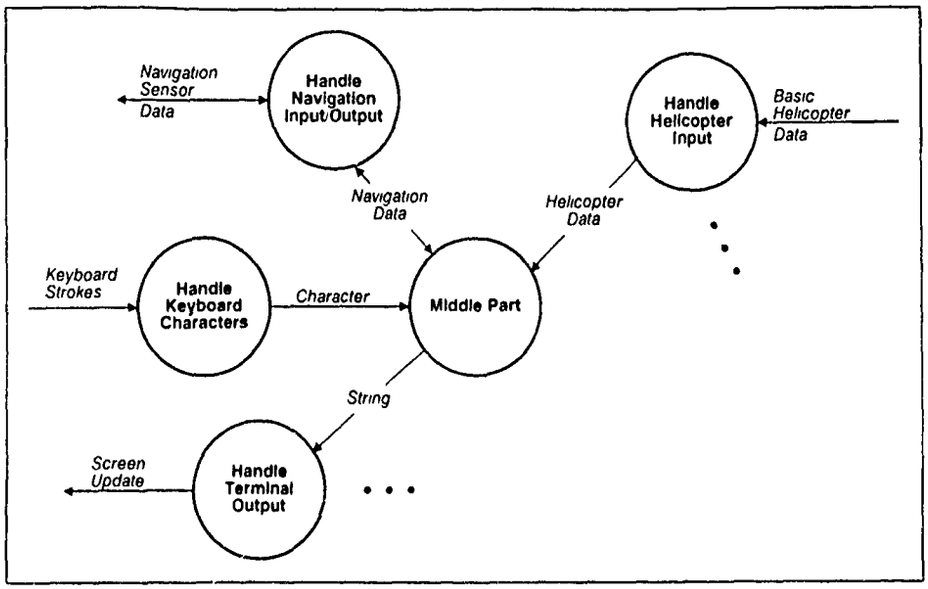


Figure 2: Top level data flow diagram

This first step is illustrated in the top-level data flow diagram (DFD). See figure 2 for a mapping of the helicopter system architecture onto a top level software DFD. The interaction with the hardware devices (navigation systems, pilot controls, ...) is controlled by concurrent handler processes. In the case of functional cohesion, some of these handler processes may be combined to one process in order to reduce the number of different tasks in the system.

In the next steps, the main avionic application (middle part) is decomposed using a hierarchy of DFD's. The tool for this step is the good old Yourdon/De Marco DFD, applied not for functional analysis of the project but for software design. Hence, one cannot just map the transforms and data flows of the requirement description onto software modules. It is necessary to set up a new, shallow leveled DFD structure with the aim to combine transforms to concurrent software processes. During this procedure, one has to consider the resulting interactions of the so-formed concurrent processes. They run in parallel and interact with each other. Therefore, one carefully has to avoid mutual waiting situations, i.e. dead-locks. Sometimes intermediary processes (buffers, queues, ...) have to be introduced to decouple applications. For the process identification, i.e. the combination of certain DFD functions, functional cohesion, as well as temporal dependencies, are to be considered. Typical reusable objects, such as monitors for a critical data region, servers or periodic modules, are examples of such processes.

Figures 3 and 4 illustrate the software DFD structure and process identification in the experimental helicopter project: Figure 3 shows that the AVT middle part consists of four major software functions (terminal, flight and graphics management, as well as moding control). For performing process identification, this level of DFD hierarchy is not sufficient. As an example, in figure 4, the flight management is further refined to a level where one can identify the concurrent processes: helicopter control, flight control, navigation and obstacle warning should run concurrently. The navigation process is a combination of the software functions navigational computation and height warning, and the data storage for NAV-data and height limits. Its real-time performance is event driven. Whenever new navigational data arrives in the system (through certain handlers, see figure 2), the process becomes active. Only in the case of new height data the height warning function will be performed. Hence, the height warning software function is integrated into the navigation process. The flight control process monitors the routes, guaranteeing mutually exclusive access to these flight routes. The obstacle warning is a typical periodic task. Every second, possible obstacles along the flight track are checked.

Now, all processes are determined and can be graphically illustrated in a single-level process structure chart. At this stage, a textual representation of the software can begin. Because of the excellent design features of Ada, this will result in readily compilable Ada PDL which corresponds directly to the graphical representation and is easily readable. The concurrent processes are described as Ada tasks. A hierarchical dependency of tasks must be avoided. All concurrency must be visible at the top level. In order to support separate compilation and readability, the tasks will be embedded in Ada packages, strictly separating the specification from the implementation (body) part. Thus, even in this early software design stage, the results can be expressed with Ada code.

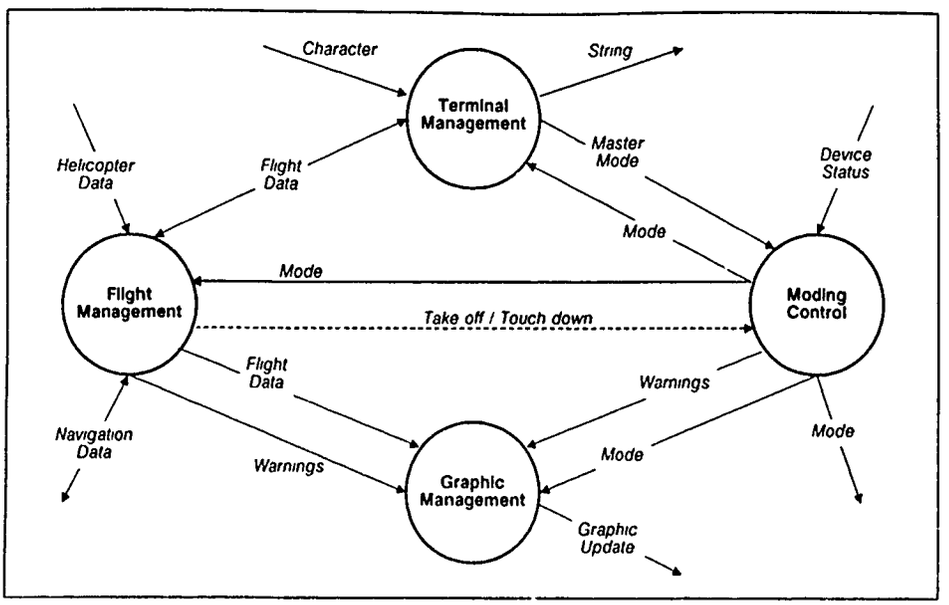


Figure 3: The middle part of the top level DFD

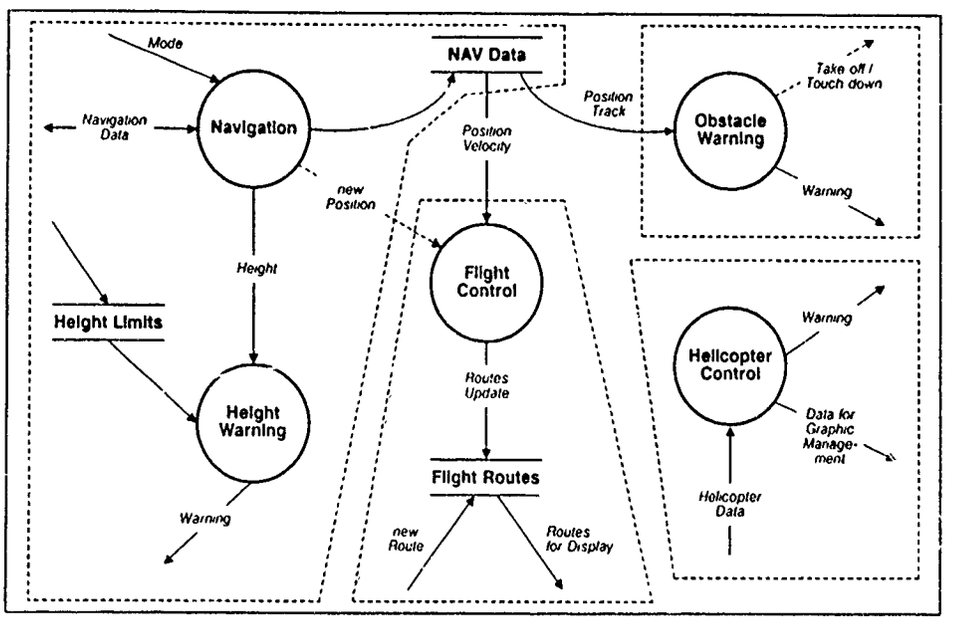


Figure 4: Refinement of the "flight management" with process identification

In order to conclude the global design, the interactions of the tasks have to be specified. The data flows between the concurrent processes are a basis for the Ada rendezvous between these tasks. In this design step, a careful decision has to be made as to which task issues an entry-call and which task is called, to avoid waiting tasks and dead-locks (caller-called decisions).

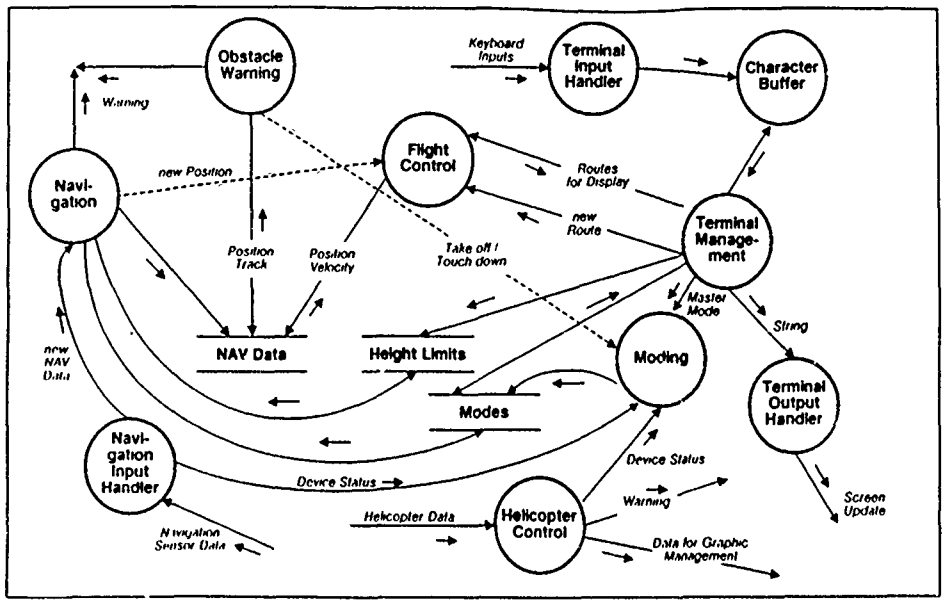


Figure 5: Ada task graph of the main CPU

The final result of such a global design is an Ada task graph (ATG) which describes the network of all concurrent processes and their interactions. In an ATG, bubbles now represent tasks, their connection arrows represent the call directions and the small arrows describe the data flows. See figure 5 for an example ATG from the helicopter project. All previously identified processes appear here on a single level (no hierarchy). The handler processes of the first design step now fit into the gradually identified processes of the main avionic part. The sometimes complicated distribution of the processes onto processors in a multiprocessor environment is beyond the scope of this paper. For an extension of this design methodology to distributed environments, see [2]. In the experimental helicopter project, the multiprocessor setup was straightforward: the main CPU (master) performs all applications while two other processors are dedicated to graphics management and report writing. In general, it should be possible to represent all processes of a single processor in a single level ATG. Otherwise, too much concurrency (too many Ada tasks) will occur. As an example of the caller-called decision, consider the interactions of the flight control and the terminal management process of figure 5: the flight control process is a pure server, i.e. is always called in order to be in an accept mode for its various jobs at any time (in Ada, this corresponds to a selective wait construct). On the other hand, the terminal management process is a pure caller. It performs lengthy computations accessing lots of data. Therefore, it always actively issues Ada entry calls. In conclusion, both interactions between the flight control and the terminal management process are called from the terminal management side, although the data flow is in one case "in" and in the other case "out".

4. Detailed design

As already mentioned, compilable, top-level Ada code can easily be deduced from the ATG by describing concurrent processes with Ada tasks and their interactions with Ada rendezvous. Performing the detailed design, the Ada package construct is applied in conjunction with OOD ideas.

In a first step, the Ada tasks are encapsulated in Ada packages. So-called "entrance procedures" are the only visible parts of the concurrent processes in the package specifications. Not until the package bodies are the tasks themselves declared and their entries identified with the entrance procedures. Hence, the tasks are treated as typical objects in an OOD manner: their detailed definition, or even their implementation is not visible. Only the operations which influence them, i.e. the entrance procedures, are visible. Figure 6 shows this OOD concept applied to the moding part of the ATG of figure 5.

In the following steps, the sequential flow of each task is further decomposed using separate Ada subprograms (which describe some DFD-transforms of the task) and again using Ada packages and procedures / functions to represent objects/operations from OOD theory. For example, the height warning (figure 4) will be implemented as a separate subprogram in the task "navigation".

Spec:

```
with DEVICES; -- Definition of devices and their status
package MODING is
    type MASTER_MODE  is (OFF, DAY, NIGHT, . . );
    type MODES        is . . .;
    function GET_MODES return MODES; -- access to data store
    -- entrance procedures:
    procedure NEW_MASTER_MODE (VALUE : in MASTER_MODE)
    . . .
end MODING;
```

Body:

```
package body MODING  is
    task MODING is
        entry NEW_MASTER_MODE (VALUE : in MASTER_MODE);
        . . .
    end MODING;
    procedure NEW_MASTER_MODE (VALUE : in MASTER_MODE) is
    begin
        MODING.NEW_MASTER_MODE (VALUE);
    end NEW_MASTER_MODE;
    task body MODING is separate;
end MODING;
```

Figure 6: An example for using OOD techniques in Ada

This concludes the discussion about the design method. The further stepwise refinement, towards a full Ada implementation is generally straightforward and heavily depends on the specifics of the application. The next chapter returns to the discussion of the most dominant feature of this design method: the real-time scheduling aspect.

5. Scheduling with the Ada tasking model

Earlier avionic projects, especially those where strict real-time requirements with short cycle times dominated, were typically realized using a global cyclical scheduler. All functional modules had to be fitted into time slots of a cyclical executive in order to guarantee their periodicity and to ensure that the critical sections of different tasks do not interleave. With larger applications, this mapping process, from a functional aspect to time slice behaviour, became more and more complicated. The average, or better the worst case duration of each function, had to be estimated to ensure that all applications fit into their time slots. An overrun would destroy the whole global execution scheme. Such an approach often confused clear program structure, violating functional cohesion and locality principles for the sake of timing considerations. Typically, such schedulers were implemented using special operating system routines or real-time extensions to the implementation language.

The Ada tasking model offers a fundamentally different approach. As observed in the previous chapters, the language itself provides all the features necessary for real-time scheduling. Whenever a task completes an execution part, has to wait for information from other tasks, or a higher priority task becomes ready to execute, the system automatically reschedules. This dynamic preemption of tasks at runtime is a direct outcome of the Ada compiler. It generates non-deterministic timelines, at odds with the very idea of the classical fixed execution time slots.

As we have observed, the design of real-time systems using Ada is guided by functional cohesion. Only those software modules whose applications are related, comprise a common task. Each task locally determines its real-time behaviour. Some execute event-driven, others periodically, locally setting up their cyclical behaviour. No global scheduler determines the system flow, only the rendezvous mechanism between the tasks guides the flow of execution. Each task schedules itself, either cyclically with the Ada constructs "delay" and "calendar.clock" or on event per rendezvous "accept" or call. Therefore, cyclical behaviour is naturally integrated with event driven processes. Overall, such an Ada design can be easily extended and with the help of the locality principle and OOD constructs many modules (packages) are reusable.

Early criticism of Ada's real-time features argued that the non-determinism of the Ada tasking model

was in contradiction to fixed real-time deadlines. But, extensive studies have proved that certain bounds on CPU utilization, in conjunction with Ada priority policies, guarantee that all tasks will meet their deadlines without knowing exactly when any given task will be running (see [3]). Without going into details of this study here, the principle ideas of the study [3] are the "rate monotonic scheduling algorithm", which gives each task a fixed priority assigning higher priorities to tasks with shorter periodicity, and the "priority ceiling protocol", which prevents dead-lock situations and unwanted priority inversions. Nevertheless, the present definition of the Ada language has some drawbacks related to the priority inversion issues (which are caused by using FIFO rather then priority queues for tasking). Hopefully, future versions of Ada, and perhaps even the next official release, Ada9X, will address this matter.

A second important criticism regarding Ada's real-time execution was the unsatisfactory quality of the Ada compilers. How big of an overhead does an Ada compiler impose on the scheduling (task switch times)? What is the accuracy of the delay and timer constructs in Ada? Benchmark tests initialized by SIGAda's performance issues working group show that the latest generation of compilers now have quite satisfactory results (see [4]). Our own experience in the experimental helicopter project was also quite satisfying. The Ada tasking model, under the rate monotonic scheduling policy, i.e. a special policy of assigning priorities to tasks, works well. Nevertheless, a few features of an Ada runtime system outside the scope of the language were used, especially to overcome the inaccuracy of the Ada timer resolution.

6. Conclusion

The experimental helicopter project, as a typical real-time system, has been completely designed and implemented with Ada. The design method derived from Nielsen / Shumate has proved to be a practical, comprehensive guideline utilizing the powerful Ada features to define and implement a complex real-time software system.

The functional decomposition served as a basis to build the Ada tasks and their rendezvous. In this process, non-cyclical modules and periodic tasks were easily combined. Hence, no painful fitting of functional applications in time slots of a cyclical executive had to be performed. The Ada compiler itself provided the real-time scheduling.

Currently, the project is undergoing a restructuring phase leading to some new or changing functionality and requirements. But, because of the well structured design and the modular, clear and reusable software implementation, due to Ada, we are convinced that we can easily cope with these new aspects.

References

[1] Kjell Nielsen, Ken Shumate
"Designing Large Real-time Systems with Ada"
McGraw-Hill, 1988

[2] Kjell Nielsen
"Ada in Distributed Real-time Systems"
McGraw-Hill, 1990

[3] L. Sha, J. B. Goodenough (SEI, CMU)
"A Review of Analytic Real-time Scheduling Theory and its Application to Ada"
Proceedings of the Ada-Europe International Conference, 1989

[4] SIGAda Performance Issues Working Group
"Ada Performance Issues"
ACM Press Ada Letters No. 3, 1990

CODE GENERATION FOR FAST DSP-BASED REAL-TIME CONTROL

by
H. Hanselmann, A. Schwarte, H Henrichfreise
dSPACE digital signal processing and control engineering GmbH
An der Schönen Aussicht 2
W-4790 Paderborn
Germany

Summary. Digital single-chip signal processors (DSP) are powerful devices to implement closed-loop controllers for highly dynamic mechanisms. Code production is however not that easy, particularly with DSP offering only fixed-point arithmetic. This paper describes key issues and a toolset which builds on automatic code generation to complement existing control design tools so as to close the gap between design and implementation or experiment.

Introduction

An integral part of many guidance and control tasks is the lower level embedded closed-loop control of mechanisms (Fig. 1).

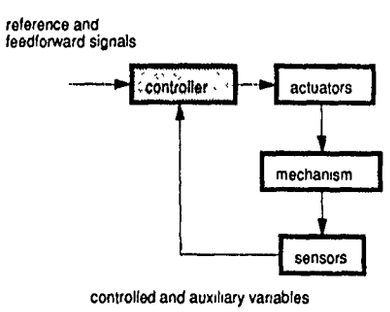


Fig. 1: closed-loop control

The mechanisms controlled may range from microminiature actuators to huge flexible space structures. Tasks include motion control, vibration damping, and stabilization. Techniques include classical multiloop PID-control, state-space control with Kalman-Filters and observers, gain-scheduling, and adaptive control.

Fixed-point and floating-point digital signal processors (DSP) are very powerful devices for implementation of such controllers for fast systems. They are also available in high-reliability versions suitable for avionics and similar applications.

Traditionally, the code for such devices is developed on the assembly language level which has well-known disadvantages. For fixed-point DSP there has been virtually no alternative. There are HLL (high level language) compilers for some DSP, but they lack adequate data types for dealing with fixed-point arithmetic other than integer. They are also not tailored to the architecture of DSP. Furthermore, producing code for such processors means more than just programming. There is much to be done between a completed controller design and the point where actual code can be produced. Crucial steps are structure selection and scaling.
For the newer floating-point DSP there are quite good C compilers and the specifics of fixed-point arithmetic no longer dominate the task of code production. More complex nonlinear control can be envisioned to be implemented fully automatically, whereas for fixed-point chips fully automatic implementation is currently limited to linear (though arbitrarily high-order) controllers, with semi-automatic treatment of nonlinearities and logic.

This paper describes key issues and a powerful commercially available toolset (DSP-CITpro) for generating code for real-time DSP-based control. This toolset has also proven to be very useful for real-time simulation (hardware-in-the-loop simulation).

DSP Chip Categories

The largest DSP family is available from Texas Instruments. This family roughly divides into three groups (Fig. 2).

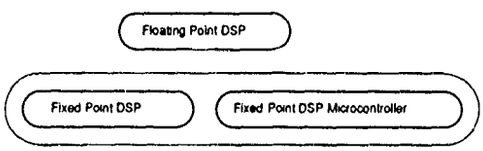


Fig. 2: DSP categories

The DSP microcontroller is just a fixed-point DSP with on-chip peripherals such as: bit i/o port, watchdog, 6 high-speed pulse-width-modulated outputs, 4 capture inputs. The latter two features belong to a so-called *event manager*, a very important subsystem which is also available in many modern non-DSP microcontrollers

The most important points of a DSP's architecture regarding code generation are:

- the type of arithmetic offered,
- the support for HLLs (high level languages),
- memory limitations.

The current floating-point DSPs are most suitable for standard HLLs such as C. They have virtually no memory limitations, software stack support, and very efficiently implement the floating-point arithmetic of standard HLLs.

Standard HLLs are not generally suited to fixed-point DSP primarily because they do not offer a suitable data type and arithmetic concept for efficient and accurate fixed-point signal processing. The DSPL language as described below is designed specifically to fill this gap. It also addresses other issues of efficient use of the DSP's special architecture and instruction set /1/

Fixed-point DSP Arithmetic

The arithmetic used throughout DSP-CITpro for fixed-point DSP is *fractional arithmetic* where the binary point is just right of the 'sign' bit (Fig. 3),

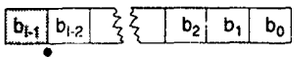


Fig 3 fractional data format

and the number range is

$$-1\,0_D \le r \le 1\,0_D - 2^{-(l-1)}$$

The central arithmetic operation in most signal processing tasks is the scalar (or dot) product

$$r = c_1 \cdot d_1 + c_2 \cdot d_2 + \ldots + c_n d_n \qquad (2)$$

It is crucial to have a clear methodology as to how this operation is to be implemented on a DSP, with respect to number range violations, accuracy, and efficiency.

Key Issues of Control Implementation

From a system dynamics viewpoint some key issues when implementing a closed-loop controller are /2/:

(1) minimization of computational load,

(2) insensitivity to coefficient and signal quantization,

(3) minimization of input/output delay,

(4) good discretization if controller prototype design is analog,

(5) adequate scaling for fixed-point arithmetic,

(6) overflow handling with fixed-point arithmetic.

From a coding viewpoint some key issues are:

(7) avoiding assembly language coding, yet exploiting processor architecture,

(8) avoiding loops, subroutines, or indexing for maximum speed,

(9) control over memory allocation,

(10) ensuring timely execution even for worst-case control flow in the program,

(11) avoiding extended precision arithmetic

Remark (8) may seem to contradict good programming practice, but need not have the unwanted effects normally associated with such programming style. For DSPL compilers as described below the level where this style becomes visible is the assembly language output, not the DSPL program. For generated C code the generated code is still easy to read.

The various points listed above will now be explained in some detail first for a linear controller. It is assumed that the controller is available in the form of matrices A, B, C, D of the time-invariant difference equation

$$x_{k+1} = Ax_k + Bu_k \qquad (1)$$
$$y_k = Cx_k + Du_k \quad .$$

where x is the internal state vector of the controller (which is assumed to have dynamics, not just gains), u comprises all input signals to the controller (i.e. from sensors or reference generators / path planning modules), and y comprises all external outputs of the controller (i.e. at least the control signals going to the actuators).

If the controller is a connection of subsystems then it is assumed for simplicity that all subsystems and connections are combined into one big 'monoblock' system (1). An example would be the connection of a state feedback gain matrix, feedforward gain matrix and a plant observer or stationary Kalman-Filter. A great deal of what is explained below would also apply if only a linear subsystem of a nonlinear controller would be considered.

(1) minimization of computational load

If matrices A, B, C, D are taken directly from a control *design* software package, then all matrices may be totally dense in the worst case. Minimization of the computational load means reducing the number of nonzero entries (coefficients) in those matrices, without altering anything in the dynamic input/output-behaviour of the controller. This can be achieved by suitable transformations based on linear algebra theory. Matrices A, B, C are affected. Totally dense matrices A, C may for example be changed into

$$A = \begin{pmatrix} 0 & 0 & 0 & & 0 & 0 & -a_0 \\ 1 & 0 & 0 & . & 0 & 0 & -a_1 \\ 0 & 1 & 0 & .. & 0 & 0 & -a_2 \\ . & & & & & & \\ . & . & & & & & \\ . & & & & . & & \\ 0 & 0 & 0 & .. & 1 & 0 & -a_{n-2} \\ 0 & 0 & 0 & .. & 0 & 1 & -a_{n-1} \end{pmatrix},$$

$$C = (0 \quad 0 \quad 0 \quad .. \quad 0 \quad 0 \quad 1) \quad .$$

The various forms of the matrices are also called *structures* due to their different block diagrams when represented graphically.

Unfortunately, such transformations may have an adverse effect on sensitivity to quantization in the processor. It is important to have tools which provide adequate structures, and can analyse sources of potential or real trouble with quantization.

(2) insensitivity to coefficient and signal quantization

Different structures in the above sense normally exhibit different sensitivity to coefficient and signal quantization. Quite frequently the structures with the least number of nonzero coefficients behave very badly in this respect. With such a structure one may be forced to use extended precision arithmetic somewhere, which quickly outweighs any gain from reducing the number of nonzero coefficients.

(3) minimization of input/output delay

If the control signal instantly depends on current sensor input samples, which is normally the case with controllers, then there is some inevitable delay between the theoretical output instant and the real one (Fig. 4).

Some of the delay may be due to A/D- or D/A-converters and is unavoidable. The delay resulting from the finite computation time in the processor however can be minimized by proper arrangement of the code (Fig. 5)

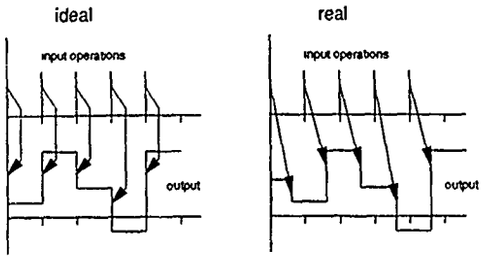


Fig. 4: input/output delay

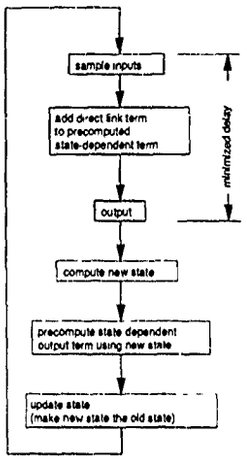


Fig. 5: input/output delay minimization

(4) good discretization if controller prototype design is analog

If control design is carried out in the continous domain (i.e. for analog implementation) the controller will normally be discretized, i.e. the differential equations will be translated into difference equations. Much can be gained by using methods which have proven to produce good discretizations. A good discretization is one which does not alter the controller's, or more importantly, the closed-loop's behaviour when compared to the analog one. Experience has shown that a good method can yield as much as a factor of five reduction in the required sampling rate over a standard discretization.

(5) adequate scaling for fixed-point arithmetic

Controller implementation on fixed-point DSP requires proper scaling of coefficients and variables. Coefficients must be representable. Variables (signals) must be scaled in order to avoid both excessive quantization for small signals and overflow for large signal excursions.

Scaling of u, y in (1) is normally derived from the gains of A/D- and D/A-converters and sensor and actuator amplifier gains. Proper scaling of x can be determined by various means. One particularly attractive method is the so-called l_1-scaling. It can be carried out by an algorithm (in DSP-CITpro) completely automatically.

Scaling of nonlinear expressions (as may be attached to an otherwise linear controller) is presently carried out manually along the same lines as known to some from analog computing, i.e. by introduction of normalized variables after determining maximum values.

Even with scaling of variables it is not guaranteed that the coefficients of scalar products are all fractional numbers. For matrices A, B in (1) this can however normally be expected with certain structures (which are often anyway the preferable ones), such as the so-called *real-modal form*. For C, D this is often not the case. The frequently high gains of a controller are represented in these matrices, and coefficients three orders of magnitude greater than one (the fractional number limit) have been experienced. Scalar-product handling as mentioned below helps.

(6) overflow handling with fixed-point arithmetic

With a scalar product (2) there is the potential of overflow. Even with proper scaling of variables such as x, y in (1) there may still be some variables which may overflow occasionally, and this may even be intentional. An example is the control signal to the actuator of a position control system. With large commanded position changes the control signal is normally expected to saturate for a while. This means that this component of y must be able to go into *saturation* overflow. The same may hold for components of x if they are associated with integrators in the controller.

Unfortunately *wrap-around* will occur if no provision for overflow saturation is made (Fig. 6).

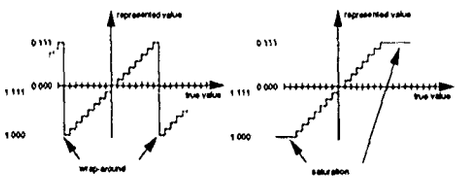


Fig. 6: overflow handling

The code generation for fixed-point DSP in DSP-CITpro via the DSPL language provides a systematic method called *scalar-product scaling* at compile time to guarantee

- accommodation of coefficients outside the fractional number range,
- efficient and accurate execution of the scalar-product operation in the extended accumulator of the DSP,
- creation of logical 'guard' bits to ensure proper saturation on overflow, if desired.

(7) avoiding assembly language coding, but still exploiting processor architecture;
(8) avoiding loops, subroutines, or indexing for maximum speed,

A. Floating-Point. For floating-point DSP, which offer 32-bit single-precision computation without speed penalty, the predominant HLL is C, but there are also compilers for other languages. For the TMS 320C30 there is even an Ada compiler available. So there is normally no need for assembly language programming except maybe for increased speed or in case of very tight memory limitations of the target hardware.

Standard HLL compilers naturally lack special constructs useful for mapping signal processing operations onto the special architecture of a DSP. Optimizations available in modern compilers can however considerably improve runtime efficiency /3/.

As an example a FIR filter operation is considered which is described by the equation below, where y is the filter output, a_i are the coefficients, and u holds the current and previously stored values of the sampled input signal.

$$y_k = a_0 u_k + a_1 u_{k-1} + \ldots + a_n u_{k-n}$$

The tasks to be performed are

- computation of the output by the scalar product, as a sequence of coefficient-times-input-variable multiplications and partial product accumulations,
- moving the stored input values u_i so as to introduce the newest input sample and discarding the oldest.

A DSP such as the TMS 320C30 can perform this operation in one cycle, although there is some setup and pipelining overhead which is significant for short filter lengths. Fig. 7 shows the assembly language code which makes use of parallel execution of 3-operand multiply and add plus address generation for both coefficients and input samples in one cycle (the code shown in the box)

```
    LDI     filter_order + 1, BK    ; block size n + 1
    LDI     AR0, address_of_last_coeffi-     , a_n
            cient
    LDI     AR1, bottom_of_sample_buffer  ; u_k-n
L   LDF     u_newest, R3
    STF     R3, *AR1++%     , u_newest -> buf
    LDF     0.0, R0
    LDF     0.0, R2
    RPTS    filter_order                  ; n
   MPYF3    *AR0++(1)%, *AR1++(1)%, R0
|| ADDF3    R0, R2, R2

    ADDF    R0, R2          ; accumulate last product
    STF     R2, y_k
    B       L
```

Fig. 7 optimal FIR filter assembly code for TMS 320C30

A problem with a HLL like C is that the compiler is not intelligent enough to detect that a piece of C code actually represents a FIR filter and could be compiled into the above code With optimizations enabled, the Texas Instruments C compiler produces code which is approximately 4 times slower than the above assembly program. The compiler is unable to generate parallel multiply and add operations (factor 2) and the 'update' operation ($u_i \rightarrow u_{i-1}$) is executed separately. This is because the 'update' must be formulated in a separate statement in C, whereas at the assembly language level a special DSP addressing mode (circular addressing) can be exploited The compiler is however intelligent enough to introduce zero-penalty looping, parallel address increment, and a delayed branch. For less structured operations the speed penalty of such a modern optimizing HLL compiler should be rather low.

In general there are some basic rules to follow for maximum efficiency, such as

(a) to avoid calculating with insignificant coefficients,

(b) to avoid loops,

(c) to avoid variable indices,

(d) to avoid pointers,

(e) to avoid unnecessary function calls

Some of these rules can normally not be met with 'general' subroutines using loops and arrays So-called 'straight-code' should be used, which is best produced by *generating* programs from higher level descriptions of the tasks. Violating these rules may result in severe degrading of execution speed. The computing power of a DSP can quickly be turned into a fraction of the peak MFLOPS rate by not letting the DSP *compute*, and doing address computation, stack administration and branching instead.

B. Fixed-Point For fixed-point DSP there are only few offerings of HLL compilers. The newer generations are sometimes supported by C compilers too. The statements on C compilers for floating-point DSP hold again, but there is one very important additional point to make: Standard C compilers have no support for doing signal processing arithmetic efficiently, i.e. there is no support for fractional number arithmetic and scalar product computation

There is one C compiler available from Analog Devices for their own line of fixed-point DSP which offers 'fractional' as a non-standard data type. But it still lacks features for optimal scalar product computation as built into DSP-CITpro's DSPL compilers It is not uncommon to see a C programming effort on a fixed-point DSP ending up in 80% (hand-written) assembly language code.

The approach taken in DSP-CITpro is to provide a suitable intermediate language, called DSPL. Details on syntax and semantics of DSPL can be found in /1/. Code examples are found below. A short characterization of DSPL and its compilers is

- emphasis on efficient and accurate fixed-point scalar product arithmetic,
- self-documentation, Ada/Pascal like syntax,
- strongly typed language,
- assembly language code is generated without loops, subroutines and address calculations for maximum speed,
- includes scalar product scaling mechanism,
- hides mechanisms for low-level operations such as details of signal i/o,
- looping, decisionmaking, and boolean instructions available,
- interrupt servicing on language level,
- compilers are processor dependent but not hardware environment dependent (i.e. target hardware may vary),
- compiler output is comprehensively commented assembly language program,
- global and statement-wise execution time profiles and memory statistics generated by compiler (see below).

DSPL compilers are available for first and second generation Texas Instruments DSPs

(10) ensure timely execution even for worst-case control flow in the program

A closed-loop controller or similar signal processing system must normally run exactly at a fixed sampling rate. It is an issue to determine the minimum execution time necessary. It is also interesting to have profile information to see which parts of the program are the most time-consuming and could possibly be improved.

The DSPL compilers automatically provide this information. For each task a global worst-case execution time is computed There are some very rare circumstances where the compiler cannot compute such information One example is a loop where the repeat count is a variable. In most cases encountered in control implementation the calculation is valid. It even takes into account that a DSP may have quite complicated instruction

cycle tables with dependencies on memory layout.

In addition the DSPL compilers also provide execution cycle information statement by statement (DSPL), embedded as comments in the generated assembly language source.

With the C compiler for the floating-point DSP there is unfortunately no such mechanism. The code must be executed for real-time profiling. A DSP-CITpro module called TRACE can however be used to gather such information from the running DSP program quite easily. A graph of the execution time history versus time can be produced, clearly showing possible fluctuations in execution time. Fluctuations occur when operations depend on the actual numerical values of operands, or if the program control flow varies. Worst-case paths in the program control flow can however only be assessed if they are actually executed.

'Executing' a program on a DSP software simulator (instruction level) could be considered an option, but is usually impractical because it is extremely time-consuming.

(11) avoiding extended precision arithmetic

A DSPs computational power can be defeated if large parts of a program require extended precision over what is provided directly by the architecture. Accumulation (such as in (2)) is normally performed already in extended precision at no penalty, but results stored for later use in the program should be at the standard wordlength.

There is one notable case in control where it may be absolutely necessary to carry out extended precision arithmetic. *High-precision motion control* may for instance require 24 bit position values on a 16-bit DSP. Using double-word arithmetic throughout the control algorithm must however be avoided. Fortunately there is a systematic way of keeping high-precision position values out of the control computation /4/, except at one easy-to-handle place. It works equally well whether the control algorithm is of simple error-driven PID type or a sophisticated optimal state variable controller including a stationary Kalman-Filter or observer.

Difference Equations or on-line Integration

In control implementation it is understood by most engineers that the algorithm has to be brought into difference equation form. Control design in the discrete domain delivers this directly. Control design in the continuous domain requires discretization. In that case it may be a viable option to implement fixed-stepsize on-line integration of the differential equations, for example with Euler or Runge-Kutta-type integrators. Advantages are:

- Parameters of the continuous system are directly reflected as program variables, and thus can easily be changed on-line, e.g for gain-scheduling.

  By contrast such parameters are normally spread out by nonlinear expressions onto the coefficients of a difference equation.

- Nonlinear parts of the differential equation are naturally represented on the right-hand side of the first-order differential equation system to be integrated.

  By contrast such parts need to be separated before discretization and be attached later, making the situation somewhat more complicated

- Sparseness of coefficient matrices of a continuous system is reproduced.

  With discretization sparseness must normally be separately generated by transforming into a suitable structure (see above section on minimization of computational load).

Disadvantages with on-line integrators are:

- Structure transformation and automatic scaling are not directly available for fixed-point DSP.
- Stability problems may occur for larger step-sizes or stiff systems.
- Integration accuracy may be more limited, and the fidelity of the digital version of the continuous system may be significantly worse than with a good discretization of the linear part.

It is worth mentioning that there are discretization methods for the linear part which correspond to an implicit on-line integrator with respect to dynamics. Such algorithms are interesting because they do not suffer from stability problems with stiff systems. Implicit on-line integrators however are totally impractical for real-time use.

On-line integration today is the first choice for real-time simulation of big nonlinear mechanical systems /5/ on floating-point DSP, where particularly the first above-mentioned advantages count.

Control Implementation with DSP-CITpro

The modules of DSP-CITpro and their interplay are depicted in Fig. 8.

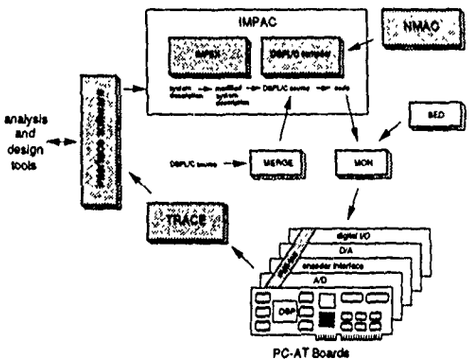


Fig. 8: DSP-CITpro modules

A brief description of the modules is now given:

Interface Software. DSP-CITpro does not cover control design or postprocessing of signals taken from the real-time experiments which may follow code generation. For these purposes there are well-known packages available such as MATRIXx or MATLAB. DSP-CITpro interfaces to these packages.

IMPAC. Impac consists of IMPEX and a C or DSPL compiler. IMPEX covers the preparation of a linear controller and analysis of implementation effects, and also generates a DSPL or C program for the controller. Such programs are then compiled by the appropriate compiler.

IMPEX. In detail IMPEX provides the following services:

- discretization,
- combination of subsystem blocks into a complete controller,
- transformation into a suitable structure,
- automatic correction for A/D- and D/A- gains,
- automatic scaling of controller states for fixed-point implementation,

- analysis of signal and coefficient quantization by simulation,
- A/D- and D/A- range entry and assignment of physical to logical i/o channels,
- code generation specifications (memory layout, scalar product scaling etc.).

The IMPEX services are available for linear blocks (1) including saturation where desired.

MERGE. MERGE is a kind of a batch editor. Its primary use is to make modifications and insertions into code which was automatically generated by IMPEX. A typical situation follows

A control design will normally be repeated and improved many times based on experiment results. Current control design technology is focusing on linear control. So in one design/implementation/experiment project a lot of versions of a linear controller are generated. Frequently the linear controller alone is not sufficient, and must be enhanced by logic or nonlinear parts. Such parts are currently beyond the code generation scope of IMPEX, so they have to be added at the language level (DSPL or C) This editing is what MERGE can automate. A suitable control file tells MERGE how to modify or enhance the input code. Manual editing can be avoided and very quick turn-around times can be achieved even if the logic and nonlinear control code finally is much larger than the code for the linear block.

NMAC. NMAC is currently only available for the fixed-point DSP. It produces DSPL-callable assembly code for nonlinear univariate functions NMAC reads a file describing arbitrarily spaced and arbitrarily many sample points of a desired function. It then produces table-lookup code which is optimized for accuracy and speed. Various parameters specified by the user can tailor the code towards speed, accuracy, and memory consumption tradeoffs

DSPL Compiler. The important features have already been discussed in the general section on code generation. More information is embedded in the examples below. The code output of the DSPL compiler is fully commented assembly code including execution time profile information. The assembly code is assembled by a standard assembler.

C Compiler. The Texas Instruments ANSI C compiler is used

MON and SED MON is an object code loader. It also loads setup information into the DSP-CITpro hardware Such setup information (A/D-ranges etc.) is bound to the object code so that loaded code and loaded setups are always consistent. SED is a setup editor. It is normally only used for static hardware setup data. Setup data for individual controllers are normally produced at the code generation stage of IMPEX.

TRACE. TRACE is the module used to record all desired variables in the DSP while the control application is running. It could be described as a virtually non-intrusive software transient recorder. Sophisticated triggering features allow capturing the relevant data. Such data can be displayed graphically or turned over to signal analysis or system identification packages (MATLAB for example)

In summary, the DSP-CITpro modules fill the gap between theoretical control design and the real experiment Turnaround times are very short due to sophisticated tools for making a controller suitable for implementation and due to code generation. It need not take more than a couple of minutes to reimplement a redesigned high-order state controller including a stationary Kalman-Filter up to and including the actual experiment.

DSP-CITpro has been used in many fields of application and for various control techniques A selection of such applications with relevance to aerospace is

- high-bandwidth suspension control for ground-based flexible structure experiment,
- lightweight compliant robot joint control,
- stabilization of head-up display mirror,
- gyro equipment control,
- servohydraulics for radar systems,
- active suspension,
- active vibration damping of flexible structures.

Control techniques involved range from simple PID control over lqg-type state controllers, observers and Kalman-Filters up to robust H_∞ controllers. Gain-scheduling, selftuning, and adaptive control can also easily be implemented even on a fixed-point DSP.

References

/1/ Hanselmann, H. and A Schwarte, "The Programming Language DSPL", Preprints/Proceedings of 1990 Power Conversion and Motion Control Conference (PCIM), Munich, June 25-28.

/2/ Hanselmann, H., "Implementation of Digital Controllers- A Survey", Automatica, Vol. 23, pp. 7-32, January 1987.

/3/ TMS320C30 Optimizing C Compiler Reference Guide, Texas Instruments, 1990.

/4/ Hanselmann, H., "Low Resolution Implementation of High-Resolution Position Control", IEEE Transactions on Automatic Control, Vol 33, No. 11, pp. 1074-1078, November 1988.

/5/ Hanselmann, H., Henrichfreise, H., Hostmann, A., and A. Schwarte, "Hardware-in-the-Loop Simulation mit Signalprozessorsystemen", Preprints Echtzeit'91 Conference (Rael-Time'91), Sindelfingen, June 11-13, 1991.

Appendix
Control Implementation Example

A motion controller for an electromechanical actuator is considered (Fig. 9). The actuator model is of 7th order due to structural mechanical resonances around 1.5 and 2 kHz. The actuator's position and the electrical driving current are measured by sensors.

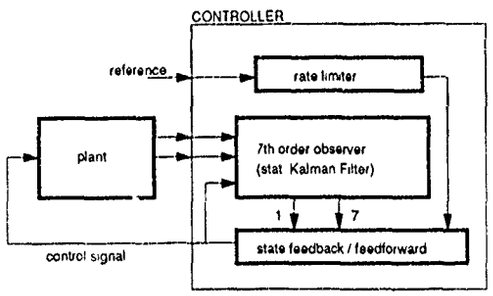


Fig. 9: controller example

The lqg-type (linear-quadratic-gaussian) controller is assumed to be designed as a linear quadratic optimal (lq) state feedback with a stationary Kalman-Filter as an observer. The design has been performed with MATLAB. A rate-limiter will be added in the reference path, making the controller nonlinear. The

rate-limiter prevents touching signal saturation at one of the sensors and thereby greatly improves medium-to-large signal behaviour.

The steps are:

(1) lqg design in MATLAB

The sampling rate is set to 10 kHz. First results within MATLAB are the state-feedback vector (including the reference feedforward gain), plus the constant Kalman-Gain matrix.

From the state-feedback and Kalman-Filter the complete controller can be constructed within MATLAB, written to disk, and converted into a format expected by IMPEX. An alternative is to write the state-feedback and Kalman-Filter to disk separately, convert them, and then use IMPEX for combining these into one single controller block.

(2) Create basic block readable by IMPEX (first alternative, excerpt only):

```
basic_block is

-- file C:\NICE\SEMC25\LQGDES\LINX.BBL
-- 20 Dec 90   1:30:54 pm

sampling_period := 1.0E-04;

system_inputs is
  name => u_x_ref, unit => V,
                   lower_bound => -1 0E+01,
                   upper_bound =>  1.0E+01;
  name => u_x, unit => V,
               lower_bound => -1.0E+01,
               upper_bound =>  1.0E+01;
  name => u_I, unit => V,
               lower_bound => -1.0E+01,
               upper_bound =>  1.0E+01;
end system_inputs;

system_outputs is
  name => u_M, unit => V,
               lower_bound => -1.0E+01,
               upper_bound =>  1.0E+01;
end system_outputs;

system_equations ssd is
  system_representation := PHYSICAL;
  system_states is
    name => x1;
    name => x2;
...
    name => x7;
  end system_states;
  dynamic_matrix is
    a( 1, 1) := -5.13340703582052E-01;
    a( 2, 1) :=  4.15508858506746E-05,
...
 row_output_matrix u_M is
    c( 1) := -3.41575795865014E+01;
    c( 2) := -1.73899220842727E+05;
    c( 3) := -2.72461533485298E+01;
    c( 4) :=  3.72279499287142E+05;
    c( 5) := -3.05696777635724E+00;
    c( 6) := -2 41915412951632E+05;
    c( 7) :=  1.36757999529863E+00;
  end row_output_matrix;
  direct_link u_x_ref to u_M is
    d :=  1.42651871551162E+01;
  end direct_link;
  direct_link u_x to u_M is
    d := -9.98336393716665E+00,
  end direct_link;
  direct_link u_I to u_M is
    d := -3.04739368990223E-01,
  end direct_link,
end system_equations;
end basic_block;
```

The above description carries all information on the controller dimension, numerical values of the matrices in (1), signal names and their optional ranges and units. This controller has a total of 80 nonzero coefficients, 49 in A, 21 in B, 7 in C, 3 in D.

(3) Application of IMPEX

The following main steps are performed:

- Structure transformation, reducing the number of coefficients in A from 49 to 11.
- Automatic state scaling for fixed-point implementation.
- Specification of i/o for code generation.
- Specification of scalar product handling as to be realized by DSPL compiler.
- DSPL source code generation.

The following DSPL program is produced (excerpt only):

```
...
scptype statel is fix'(acculength => 32,
                       round => on,
                       scale => on,
                       saturation => off),
scptype outl is fix'(acculength => 32,
                     round => on,
                     scale => common,
                     saturation => on);
...
xk   : vector (7) of fractional;
xkl  : vector (7) of fractional;
u    : vector (3) of fractional;
input is u;
y    : vector (1) of fractional;
output is y,
templ : rawaccumulator;
...
begin
  every  1.0E-04 do
    update (xkl, xk);
    input (u);
    accumulate prescalpro (outl)
      y(1) := templ + dl * u;
    end accumulate;
    output (y);                          -- line 152
    accumulate scalpro (statel)
      xkl(1) := al * xk + bl * u;
    end accumulate;
...
```

(4) Compilation and download (first without rate-limiter)

The linear controller's DSPL code is first compiled for a TMS 320C25 second generation fixed-point DSP, the assembly language output assembled, and the object file loaded onto the hardware if it is attached. All that can be done by invoking just one batch file.

Compilation yields the following processor load info file, which shows that at maximum 27.6 µs are needed:

```
execution time requirements

task | cycles | rate (kHz) | time (us) | rqst (us)
-------------------------------------------------------
   1 |    276 |     36.232 |    27.600 |   100.000

total processor load     27.60 %

  303 words of code (off-chip).
   37 words of data (on-chip).
   32 words stack   (on-chip)
```

An excerpt of the assembly code produced highlights the automatically generated comments and DSPL statement-wise execution cycle profile information. Line 152 of the DSPL source is marked in the above DSPL excerpt for reference.

```
...
       addh  _c15
       rovm                      ; disable HW satur
       sfl                       ; rescale
       sach  _v2, 7              ; y(1)
;
; ---- 19 cycles
;
; line 152
       ds2101 0,2,_v2,080h,2     ; output y(1)
;
; ---- 7 cycles
;
; line 153
       zac
       it    _v8                 ; xk(1)
       mpyk  3172                ; a1(1)
       lta   _v8+1               ; xk(2)
       mpy   _c3                 ; a1(2)
       lta   _v1                 ; u(1)
       mpyk  -76                 ; b1(1)
       lta   _v1+1               ; u(2)
       mpyk  -3660               ; b1(2)
       lta   _v1+2               ; u(3)
       mpyk  -23                 ; b1(3)
       apac
       adlk  1, 14 - 0           ; perform rounding
; no overflow test, rescaling 0 bit
       sach  _v9, 1              , xk1(1)
,
, ---- 15 cycles
,
...
```

(5) Merging-in the rate-limiter code

The following file instructs MERGE so as to include the rate-limiter code:

```
@ #begin# - 1 insert
#
r_last : fractional,
delta : fractional,
max_delta : constant fractional := 0.008;#

@ #update# +1 insert  #     r_last := u(1);#

@ #accumulate# insert
#--------------------------
    delta := u(1)-r_last,
    if delta > max_delta then
      u(1) := r_last + max_delta;
    elsif -delta > max_delta then
      u(1) := r_last - max_delta;
    end if;
--------------------------#
```

The first line of this merge control file for instance contains the following merge instructions: look for the string 'begin', go up one line, then insert the 3 lines of declarations bracketed in by the # character. One executable statement and a sequence of 6 executable statements are inserted by the next two merge instructions.

Invoking MERGE for the above DSPL source and merge control file, compiling, assembling and download can again all be performed by just invoking one batch file.

(6) C code generation

If the same controller is to be implemented on the TMS 320C30 floating-point DSP C code generation has to be selected in IMPEX. The same MERGE utility can then be used to insert the rate-limiter. An excerpt of the resulting code is shown below:

```
/* declaration of input / output functions */
void start();
float ds2001();
void ds2101();
/* declaration of coefficients */
/* dynamic matrix */
float a1_1 =  9.6813219E-02;
float a1_2 =  8.8771683E-01;
...

/* input matrix */
float b1_1 = -1.4816430E+00;
...

/* declaration of variables */
/* state variables */
float x1_modal =  0.0000000E+00;
float xk1_1 =  0.0000000E+00;
...

/* input variables */
float u_x_ref_scaled =  0.0000000E+00;
float u_x_scaled =  0.0000000E+00;
float u_I_scaled =  0.0000000E+00;
/* output variables */
float u_M_scaled =  0.0000000E+00;
/* temporary variables */
float temp_1 =  0.0000000E+00;
/*--- rate limiter ---------*/
float r_last,
float delta,
float max_delta = 0.008;

/*--- exec time ---------*/
long timer_new, timer_old;
long *timer_counter = (long *) 0x808034;
float exec_time;
float t_clock = 1.2012E-7;

c_int10()
{

  timer_old = *timer_counter,
  asm("trapu 27"); /* call TRACE30 */
  x1_modal = xk1_1;
  x2_modal = xk1_2;
  x3_modal = xk1_3,
  x4_modal = xk1_4;
  x5_modal = xk1_5;
  x6_modal = xk1_6;
  x7_modal = xk1_7;
/*--- rate limiter ---------*/
  r_last = u_x_ref_scaled;
/*--------------------------*/
  start();
  u_x_ref_scaled = ds2001(0x00000000, 0x00000001);
/*--- rate limiter ---------*/
  delta = u_x_ref_scaled-r_last;
  if (delta > max_delta)
    u_x_ref_scaled = r_last + max_delta;
  else if (-delta > max_delta)
    u_x_ref_scaled = r_last - max_delta,
/*--------------------------*/
  u_x_scaled = ds2001(0x00000000, 0x00000002);
  u_I_scaled = ds2001(0x00000000, 0x00000003),
  u_M_scaled =
    temp_1 +
    d1_1 * u_x_ref_scaled +
    d1_2 * u_x_scaled +
    d1_3 * u_I_scaled;
```

```
ds2101(0x00000080, 0x00000001, u_M_scaled);
xk1_1 =
  a1_1 * x1_modal +
  a1_2 * x2_modal +
  b1_1 * u_x_ref_scaled +
  b1_2 * u_x_scaled +
  b1_3 * u_I_scaled;
xk1_2 =
  a2_1 * x1_modal +
  a2_2 * x2_modal +
  b2_1 * u_x_ref_scaled +
  b2_2 * u_x_scaled +
  b2_3 * u_I_scaled;
xk1_3 =
  a3_3 * x3_modal +
  a3_4 * x4_modal +
  b3_1 * u_x_ref_scaled +
  b3_2 * u_x_scaled +
  b3_3 * u_I_scaled;
xk1_4 =
  a4_3 * x3_modal +
  a4_4 * x4_modal +
  b4_1 * u_x_ref_scaled +
  b4_2 * u_x_scaled +
  b4_3 * u_I_scaled;
xk1_5 =
  a5_5 * x5_modal +
  b5_1 * u_x_ref_scaled +
  b5_2 * u_x_scaled +
  b5_3 * u_I_scaled,
xk1_6 =
  a6_6 * x6_modal +
  b6_1 * u_x_ref_scaled +
  b6_2 * u_x_scaled +
  b6_3 * u_I_scaled;
xk1_7 =
  a7_7 * x7_modal +
  b7_1 * u_x_ref_scaled +
  b7_2 * u_x_scaled +
  b7_3 * u_I_scaled,
temp_1 =
  c1_1 * xk1_1 +
  c1_2 * xk1_2 +
  c1_3 * xk1_3 +
  c1_4 * xk1_4 +
  c1_5 * xk1_5 +
  c1_6 * xk1_6 +
  c1_7 * xk1_7;
timer_new = *timer_counter;
exec_time = (timer_new - timer_old) * t_clock;
```

The C code as generated by IMPEX has been enhanced via MERGE with some code for execution time measurement by TRACE.

The code exhibits indexless (no-arrays) calculation for maximum speed The dsXXXX functions are i/o functions for the specific hardware.

A comparison of 1st, 2nd and 3rd generation DSPs executing the above controller including the rate-limiter programmed (or better. code generated) in the appropriate language is given in the table below. The execution time figures are exclusive i/o, which can take about a microsecond with the right peripheral hardware architecture and components

processor	language	clock	execution	
TMS 320C14	DSPL	25 MHz	26 6 µs	
TMS 320C25	DSPL	40 MHz	18.1 µs	
TMS 320C30	C	33 MHz	≈ 15 µs	about 5 MFLOPS

COMPUTER AIDED DESIGN OF WEAPON SYSTEM GUIDANCE AND CONTROL WITH PREDICTIVE FUNCTIONAL CONTROL TECHNIQUE

Didier CUADRADO,
Philippe GUERCHET

THOMSON-CSF
Division Systèmes Electroniques
9, rue des Mathurins
F-92223 BAGNEUX CEDEX

S. ABU EL ATA DOSS

ADERSA
7, Bd du Maréchal Juin
F-91370 VERRIERE-LE-BUISSON

ABSTRACT.

Predictive Functional Control (P.F.C.), a Mode Based Predictive Control (MBPC) technique is a control strategy based on the use of a model to predict the process output over a long range time period. This technique, fully compatible with the "CAD-based integrated design", is presented here. The link between the specification and the control law tuning parameters is made and the benefits of the use of a CAD tool is demonstrated. Two industrial applications are detailled. The first one concerns the guidance law of an air defence short range missile. The second one consists in the control of the two axis turret of a very short range air defence weapon system.

1. INTRODUCTION

Design of control laws for weapon systems are subject to more and more severe constraints. These laws have to satisfy high quality performance and have to be adapted to more and more difficult environments (inter-changeability of sensors and/or actuators, restricted and/or evolutive specifications, processes with unusual behavior : delay, non-minimum phase, non-linear, non-stationary, oscillatory, unstable, ...etc). In these conditions, tight correlation between the parameters to be tuned and the required specifications have to be established in order to allow rapid prototyping. A good performance/cost ratio is thus necessary.

Predictive Functional Control (PFC) technique presents interesting characteristics for this purpose. PFC is a model based predictive control technique developed by ADERSA and used by THOMSON-CSF in the last few years. The control is designed according to a receding horizon strategy, using explicitly a model to predict the process output over a long-range time period. For linear models, this leads analytically to a linear regulator which can be easily implemented in an on-board computer for real-time applications.

The technical features of PFC are well appreciated : follow-up servo performances, robustness, simple constraints handling, possibility of controlling difficult processes, compensation of measured disturbances by a feedforward action, inherent dead-time compensation, ... But the most appreciated characteristic is its ease of tuning ; in fact PFC design introduces specification parameters rather than tuning parameters this allowing direct relation with performance. This main feature makes the PFC technique fully compatible with the "CAD-based integrated design" and "engineering workstation" concepts.

PFC environment permits acceleration of the phase between model identification and performance evaluation. Once the model is obtained, the control strategy is straight-forward by defining the setpoints nature and the required specifications. In the PFC software, many procedures are dedicated to assist the user in the parameters selection in relation with the specifications. Simulation and behavior analysis (in both time and frequency domains) of the process controlled by PFC can also be accomplished by the software.

An expert system has been added by THOMSON-CSF to the PFC software in order to save experience of existing PFC users and to help for rapid training of new personal. The PFC concepts can thus be mastered in a short time ; this is particularly attractive for non-expert technical persons who have a limited control background.

All the tools associated with the PFC software increase considerably the interest in the technique. PFC is wide-open to further extensions.

The main characteristics of the PFC software are given in section 2.

In the fields of air defence systems, in which we apply it, this technique present moreover the advantage, with regard to conventional techniques, to propose a package :

- complete for the computation of the control (it includes both feedback and feedforward actions in the case of unknown setpoint),
- independent in the case of temporary lack of measures (setpoint or output process).

This technique has been used by THOMSON-CSF mainly in the following two applications :

- in simulation (engineering simulator of short-range weapon system) for all phases of line-of-sight guidance (initial, pursuit and terminal) of a high velocity missile. The missile characteristics are : non-stationary, non-linear and non-minimum phase. This technique has proved to be very performant with regard to a classic law (adaptative PID) in the case of fast manoeuvring targets (aircraft evasive manoeuvres, missile and aircraft helicoidal manoeuvres) and it maintains very good performances in the case of straight targets, radial or not.
- in simulation and on the actual weapon system for the turret homing to target phase of a very short range weapon system. This turret presents many non-linearities such as friction, free motion, hysteresis and saturation. Its behavior depends on the amplitude of the effective movement. The PFC technique with its associated CAD tool, allowed, not only to improve the dynamic performance with regard to polynomials regulators (previously used) but also rejects the effect of perturbations such as blast of wind, noises and inertia variations. The looked for robustness was not, in the past, achieved by other technique.

These two applications are treated in section 3.

2. PFC SOFTWARE

The general principles of the PFC technique are briefly presented here, in the single input - single output case, then the main characteristics of the software are described and its CAD-compatible nature is shown. Details on the PFC algorithm which leads to a linear control expression are provided in the Appendix A.

2.1. PFC GENERAL PRINCIPLES

PFC is a particular long-range predictive control technique. The control variable is calculated on-line according to the following receding horizon strategy.

At each sampling time (Figure 1) :

- the process output has to rally a setpoint trajectory in the future.
- a reference trajectory initialized on the actual process output defines the way to follow the setpoint on a prediction horizon ; the characteristics of this trajectory are chosen in relation to the desired closed-loop dynamics. A first order decay error between the setpoint and the process output is generally used.

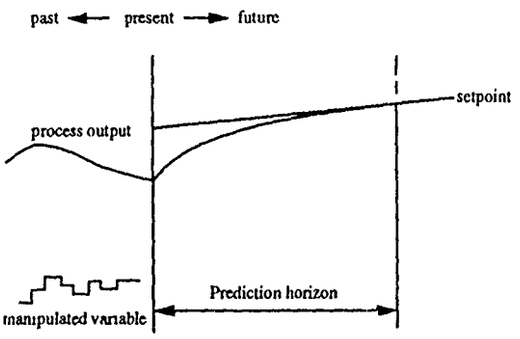


FIGURE 1 : PFC principle

- The future control variable is structured as a linear combination of a pre-specified set of functions called base functions. The choice of these functions depends on the nature of the process and the setpoint. Gererally step, ramp, parabola, ... are used.
- The process model allows expression of the output prediction under the effect of the future control sequence. The process output prediction is then adjusted by taking into account the extrapolated distance (in the future) between the process and model outputs based on past observation. This is called the self-compensation procedure.
- The control objective is to minimize the sum of the squared errors between the predicted output and the reference trajectory at certain points of the prediction horizon called the coincidence points. The number of these points is at least equal to that of the base functions. Adding a quadratic term in the control variable or its variations to this criterion allows control smoothing.
- The control variable computation consists of determining the unknown coefficients of the linear combination of the control expression. Only the first value of the future control sequence is used to control the process. The whole procedure is repeated at the next sampling instant and so on.

2.2. PFC PARAMETERS AND THEIR PERFORMANCE RELATION

It is assumed here that the reader is already acquainted with the Appendix 1 in which details on the PFC algorithm are given. We first present the list of the PFC tuning parameters and then we indicate how they can be selected.

The parameters are of two types : basic and optional :

- basic parameters, these are :
  - . the base functions
  - . the reference trajectory (response time)
  - . the coincidence points

  The influence of these basic parameters on the main specifications is illustrated in the following table where 0,1 and 2 mean weak, medium and high influence respectively

Tuning / Specification	Base functions	Reference trajectory	Coincidence points
Steady-state accuracy	2	0	0
Closed-loop dynamics	0	2	1
Stability - robustness	0	1	2

  The almost diagonal property of this matrix shows that the basic parameters are specifications parameter directly connected to the performance characteristics.

- Optional parameters, these are for :
  - . self-compensation (extrapolation polynomial degree, number of past observations and their filtering)
  - . setpoint extrapolation (degree of the polynomial, number of past values)
  - . criterion modification (weight of the quadratic term added and order of the control variable variation considered)

  Concerning the influence of these optional parameters, we can give the following elements.

  The self-compensation procedure is necessary when the difference between the process and model output causes a not constant asymptotic tracking error if no self-compensation is used.

  The setpoint extrapolation in a polynomial form ensures the steady-state accuracy when the setpoint is not known in the future.

The criterion modification is intented to produce a more regular control. It is particularly interesting in the presence of noise.

2.3. TUNING

For some of the already listed PFC parameters the specifications can guide their selection, this is the case of the time response of the reference trajectory and the setpoint extrapolation degree.

For some others, rules derived from theoretical results have to be applied ; this is the case for the choice of the base functions.

Unfortunately, for the remaining parameters, there is no straight forward link between them and the performance characteristics. Thus for the selection of these parameters, no rules exist apart from some elementary ones. For these parameters the software PFC provides many help procedures to the user, through specific experimental results giving, for different choices of the parameters, many performance criteria (closed loop dynamics and robustness margins, ...).

The limited scope of the paper does not allow to give all the details concerning the tuning of each parameters.

But one can be said is that the possibilities offered by the PFC software permits easy tuning and rapid prototyping. PFC can be used by non-experts, it is a good candidate for CAD-based Integrated Design.

2.4. TUNING WITH AN EXPERT SYSTEM

An expert system, called PFC-EXPERT, has been built from KIRK (developed by THOMSON-CSF). KIRK is an expert system shell. Its principal functionalities are a forward chaining capability, a prolog-based backward chaining capability, packet of rules to organize the knowledge base.

The PFC-EXPERT knowledge base contains the tuning rules related to basic parameters (see section 2.3). In the case of optional parameter, the rules are defined from experience of existing PFC users acquired by interpretation of performance criteria provided by the software PFC.

The PFC-EXPERT presente advantages to accumulate the knowledge of the users and to help for rapid training of new personal.

3. APPLICATIONS

3.1. MISSILE GUIDANCE

3.1.1. PROBLEM STATEMENT

Interception of a moving target (aircraft or missile) by a remote control missile, launched from a fixed or moving platform, is a complex dynamic problem composed of several phases. Once the target is detected, the launching platform is oriented such that initial conditions of the missile flight are the most favorable. Simultaneously, the weapon system computer estimates the most appropriate time to launch the missile. As soon as the missile is launched, it is guided until the interception of the target. In the domain of short-range weapon systems, the type of missile guidance is generally a L.O.S. guidance (Line of Sight).

The L.O.S. concept ban be characterized by three points : the position of the fire unit, of the target and of the missile. The object of the L.O.S. guidance system is to constrain the missile to lie as nearly as possible on the line joining the fire unit and the target called the line of sight.

The concept of THOMSON-CSF L.O.S. systems is automatic using differential missile to target tracker (ex. CROTALE system) where all the operations are executed at ground level to correct imperfect target tracking in the guidance loop.

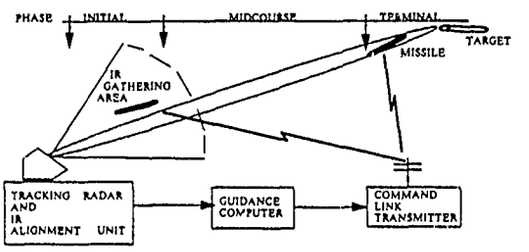


FIGURE 2 : Principle of the LOS command

The main research works on the missile guidance subject [2], [3], [4] concern the "terminal" phase (see figure 2) because it is a deciding factor in the presence of targets manoeuvres, particularly during the hundredths of second preceding the interception. Nevertheless, each phase of the missile trajectory has a different objective which leads to the success of the fire. The first phase is intented to counter the launch disturbances. The second phase objective is to minimize the energy consumption to preserve a high potential of manoeuvrability. Finally the "terminal" phase is concerned with the miss distance minimization. Generally, there exists a guidance law structure adapted to each phase and an appropriate methodology for tuning the parameters of each guidance law.

In the most general case, the inputs of the guidance law are (see figure 3) :

- ε_{tm}, measures of the metric differential gap target to missile,
- θ_a et $d\theta_a/dt$ measures of position and speed binded to the line of sight motion,
- r_m, estimated value of fire unit to missile range.

The acceleration Γ_c (see figure 3) is the output of the control computer and is calculated in the missile reference system of coordinates.

The specifications required for the guidance law are :

- to minimize the error ε_{tm} at the interception, particularly in the case of fast manoeuvring targets,
- to minimize the noises on the missile control accelerations,
- robustness with respect to the inaccurate knowledge of the process (non stationary, non linear, non-minimum phase, reference system change, ...etc).

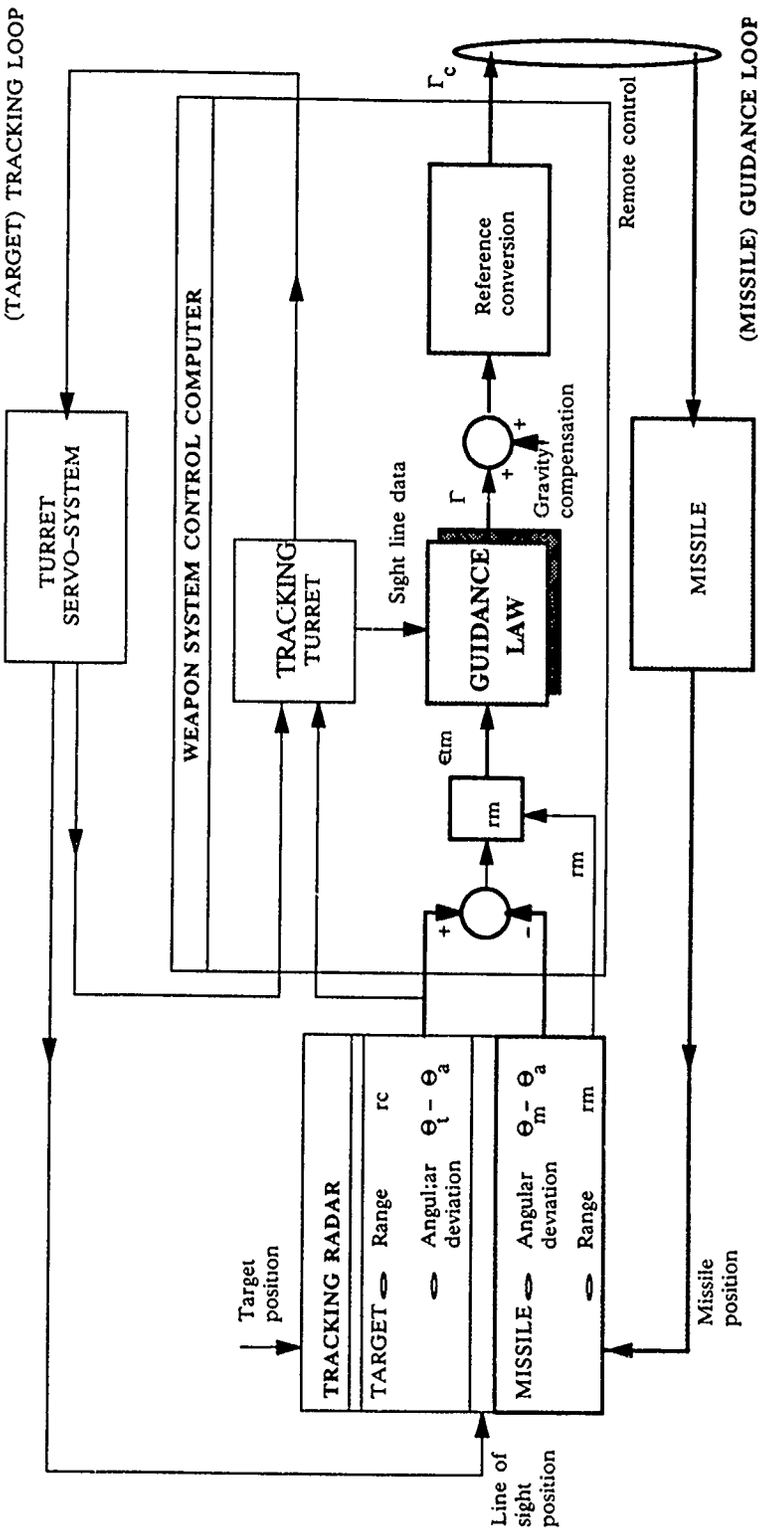


FIGURE 3 : Structure of guidance law

3.1.2. CONTROL LAW DETERMINATION WITH CAD TOOLS

The following elements are needed for the predictive guidance law :

- the calculation of the future setpoint trajectory. The more convenient way to do that, is to extrapolate filtered past values in a fixed reference system,
- the calculation of the position of the missile in the selected reference system axes. It can be easily reconstructed from the mesured error ε_{tm} and the estimated setpoint (see figure 4 below),
- a linear model that gives the realized position of the missile from any control acceleration.

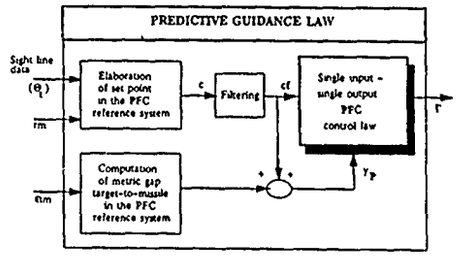


FIGURE 4 : Structure of the predictive guidance law

The single input - single output PFC control law elements are described in figure 5 :

Where : n = current time,
t_{tg} = time to go, estimation of the remaining time before interception,
tmd = output of a target manoeuvre detector.

The internal model chosen is relatively simple in comparison with the process complexity and does not result from a very accurate identification procedure. It consists in a transfer function which represents very approximatively the missile for a given flight time, a total time delay in the guidance loop and a double integration. The parameters are considered time invariant.

In what follows, two PFC laws are described by mean of the same set of equation. "PFC terminal" includes a terminal phase with the time to go estimation t_{tg} and "PFC" is the same with t_{tg} = 0 assumption. The passing over from "PFC" law to "PFC terminal" law is very easy.

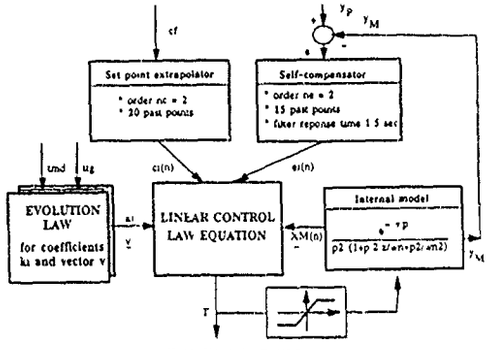


FIGURE 5 : PFC law

The principle of "PFC terminal" law is to vary simultaneously the coefficients (k_i(i=0,1,2), v) of the PFC regulator in function of two variables : the output tmd of the target manoeuvre detector and the estimation of the remaining time before the interception (t_{tg}). This leads to the following regulator equation :

$$\Gamma(n) = k_0(tmd, t_{tg}).(c_0(n) - y_p(n)) + \sum_{i=1}^{2}[k_i(tmd, t_{tg}).(c_i(n) - e_i(n))] + v^T(tmd, t_{tg}).x_M(n)$$

The evolution expressions for coefficients k_i and v are given by :

$k_i = k_{1,init} * (a_i - b_i*tmd) * cvk_i$ for i = 0, 1, 2
$v = v_{init} * (a_1 - b_1*tmd) * cvk_1$

where :

- a_i, b_i and cvk_i are positive,
- the values $k_{i,init}$ (i = 0, 1, 2) and v_{init} are those corresponding to a tuning which requires a weak dynamics (i.e. that the coincidence points are nearly of the reponse time).
- for tmd : a law value is adapted to helicoidal and evasive manoeuvres requiring strong dynamics, a high value is adapted to defiling targets which need medium dynamics.
- $cvk_i = 1. + ck_i*cvk$ for i = 0, 1, 2 where cvk is a limited function of $1/t_{tg}$.

The numerical values of a_i, b_i, vk_i for i = 0, 1, 2 result, with the PFC software, from analyzing the evolution of the coefficients k_i (i = 0, 1, 2) and v when the coincidence points are chosen smaller and smaller. The PFC software offers very great facilities for that. These variations are linear functions of tmd and t_{tg}.

3.1.3. PERFORMANCES COMPARISONS

The 6 DOF (degree of freedom) simulator used to evaluate the performances of the different guidance laws, is representative of an existing short-range weapon system. It encloses an accurate modelisation (physical models) of each of its material components. The on-line software simulation of the weapon system is according with the real time implementation. The conditions in which the guidance laws have been tested are thus among the more realistic ones.

The fire topics are

- R1 : aircraft target, radial straight target, constant speed = 500 m/s, interception range = 6.5 km,
- H1 : aircraft target, helicoidal manoeuvring target, constant speed = 300 m/s, around Δ parallel to X with a 500 m cross range, interception range = 7.4 km
- H2 : same as H1 with missile target and constant speed = 160 m/s
- D1 : aircraft target, straight and level a long Δ altitude = 1000 m, horizontal cross range = 4500 m, constant speed = 350 m/s, interception range = 4600 m.
- D2 : same as D1 with constant speed = 500 m/s, Δ altitude = 200 m, cross range = 3000 m, interception range = 4200 m
- E1 : aircraft target, straight horizontal target, constant speed = 500 m/s, escape manoeuvre 5 g before interception range 5 km.
- E2 : same as E1 with 10 g manoeuvre.

In what follows, we present synthetic results of the Monte-Carlo simulations for the different laws evaluated : "classical" law ("PID" type adaptative regulator with feedforward action), "PFC" law and "PFC terminal" law.

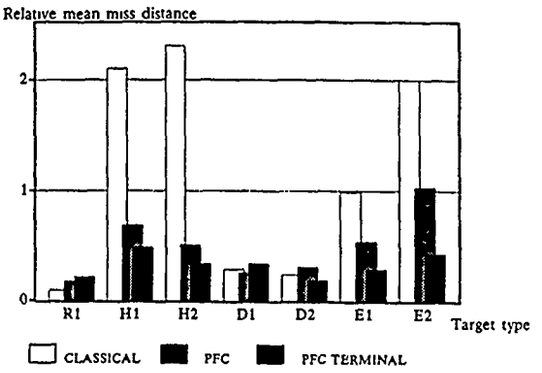


FIGURE 6 : Guidance laws comparisons

The superiority of the predictive guidance law is obvious in the case of fast manoeuvring targets. It is clear that the adding of a terminal phase improves the performances in term of miss distance from 30 to 60 %. For the last laws and from a real time point of view, realistic estimations of the duration demonstrate the compliance with the on board computer constraints.

3.2. TOW AXIS TURRET CONTROL

3.2.1. PROBLEM PRESENTATION

The objet of this application is to realize the control, using a real time computer, of a two axis (elevation and bearing) turret. The system allows the angular tracking of a manoeuvring aerial target.

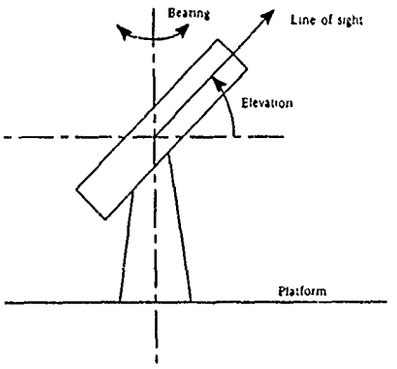


FIGURE 7 : Two axis optical turret

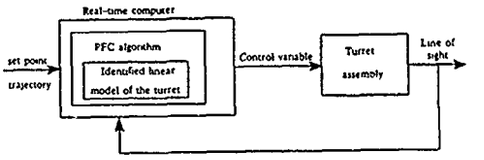


FIGURE 8 : Functional scheme of the turret control

The system includes electromechanical structures and electronic parts. It can be represented by a three dumb-bells model. The main mechanical components of the chain are the motor, the gear box assembly and the load. The chain caracteristics are :

- the motor, gear box assembly and load inertias
- the flexibility and the damping of the transmission gear
- the free motions, frictions and hysteresis
- the gear ratio

Electronic loop are included in the turret assembly hardware :

- regulation loop for the motor current
- tachometer loop for regulation of the motor speed

Synchro-resolvers give the angular positions (elevation and bearing) of the load compared with the platform.

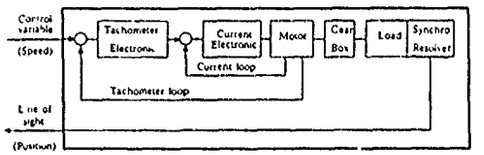


FIGURE 9 : Turret assembly (one axis)

The control variable of the turret assembly chain is representative of the load speed. The process output are the angular positions of the load which represent the line of sight for the tracking.

Now let us talk about constraints and required performances.

The most important constraint is that the control law has to withstand :

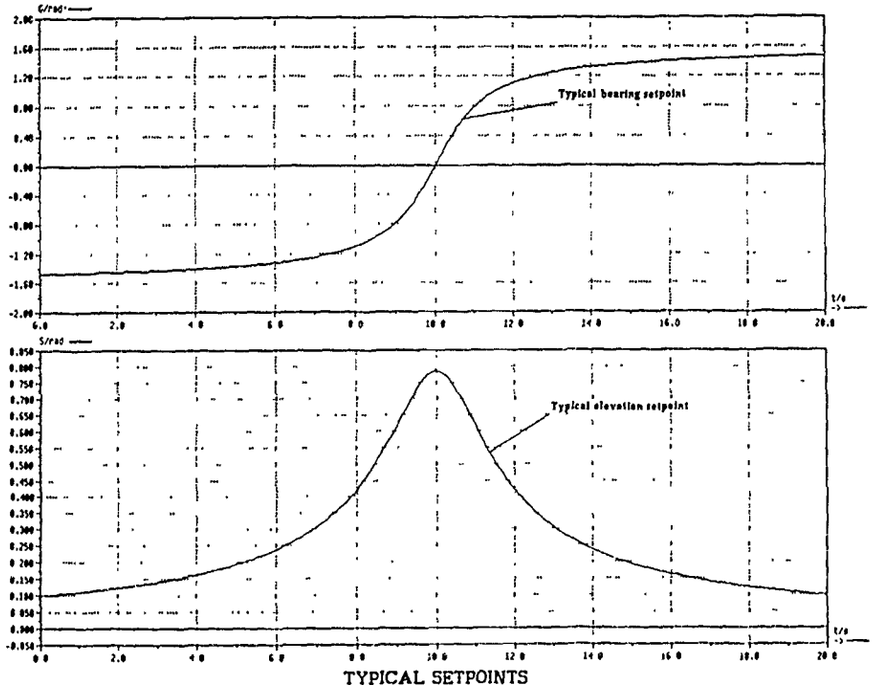
- large system parameters variations (for example : the range of the turret inertia is from 60 to 120 $kg.m^2$). These variations have an effect on the process time response (30 %) and on the process time delay (50 %)
- non linear effects due to friction, threshold, backlash...

For a tracked flying target, the set point trajectories are unknown. Nevertheless, most of them correspond to a rectilinear uniform flight at a constant altitude. The involved setpoints are made of reversed trigonometric functions (see figure 10) :

For example : $\theta = \text{arctg}\frac{Vt}{d}$

where V is the target speed
d is the crossing range
t is the current time
θ is the bearing setpoint

Morever, the turring parameters are the system required specifications. This feature makes the PFC (the choosen MBPC) technique fully compatible with the "CAD based integrated design" and "engineering workstation" concepts.



TYPICAL SETPOINTS

FIGURE 10 : Typical setpoints

The allowed tracking error in the previous conditions should not exceed 5 mrad with a rallying time response as shorter as possible.

In order to preserve the electromechanical parts, the control is subject to the following constraints :

$$| u |_{max} \leq 1.5 \text{ rad/s and } \left| \frac{du}{dt} \right|_{max} \leq 1.6 \text{ rad/s}^2$$

As the real time computer is not only dedicated to the tunet control, the sampling time cannot be less than 20 ms, which is a severe restriction according to the system time response

As all these constraints and required performances are very strict, a conventionnal control law (Signal Based Control like P.I.D.) has revealed itself quite insufficient because :

- the set points to follow without tracking error look like second order polynomials,
- the control variable constraints are very strict and are often hit (the model evolves with constrained control),
- the sampling time (20 ms) is very important with respect to the process time response (35 ms).
- the process is highly non linear (friction is an important constraint) and a SISC control law would not achieve the required robustness performances.

A more elaborate control law is needed. Among the Model Based Control techniques, the Predictive ones (MBPC) are particularly well adapted to high evolutive setpoints.

3.2.2. CONTROL LAW DETERMINATION

All the PFC parameters are listed in section 2.2. For same of them, the specifications can actually guide their selection. For example, the setpoint extrapolator degree is linked to the typical observed setpoints most of which can be approximated by second order polynomials.

The time response of the reference trajectory is related to the process response. As the control law and its gradient are limited, the time response of the reference trajectory depends on the difference between the setpoint and the process output which is to be compensated. Then this PFC parameter has been tabulated according to the error to compensate.

$$t_r = f(c, s_p)$$

Others parameters tuning derives from theoretical results. It is the case for the number of base functions. If d is the degree of the polynomial setpoints and n_i is the number of integrators in the process to control then the number of base functions is $d - n_i + 1$ if the specification is to have no tracking error. For this application, as the process is once integrative and as typical setpoints look like second order polynomials, two base functions were chosen : the step and the ramp.

For an equation system resolution reason, the number of coincidence points must at least be equal to the number of base functions. If it is greater, the system is solved with a least square method. Their positionning depends on the desired robustness. It also has some effect on the control dynamics. Situated near the current instant, the band pass is large, far away, the dynamic is quite poor.

Unfortunatly neither process behaviour specifications nor theoretical results can be a way to determine the number of coincidence points and their location.

Here is one of the main interest of the CAD tool. As soon as the reference trajectory, the base functions and the number of coincidence points are chosen, an on-line help gives to the user the gain margin, the delay margin and the time response for different locations of coincidence points. Results are presented in a table in which the user can choose a tuning for the coincidence points. An example of one of these on-line help tables is given hereafter (figure 11) for a first order reference trajectory (t_r = 0,1 s), two base functions and two coincidences points.

*No	points coincidence	k0	lm.gain	lm retard	tps. rep.
* 1	0 0400	0 1000	45.3	6 0	0 0600	0 120 *
* 2	0 0600	0.1000	33 4	6.8	0 0600	0 140 *
* 3	0 0800	0.1000	27.7	7.4	0.0800	0 140 *
* 4	0 0400	0 0800	47.2	5.8	0.0600	0.120 *
* 5	0 0600	0 0800	36 1	6 6	0 0600	0 120 *
* 6	0,0400	0.0600	50 5	5 8	0.0600	0 120 *

FIGURE 11 : Example of coincidence points on-line help results

As already told above, most of the setpoints can be considered as portions of second order polynomial. This gives the degree of the setpoint extrapolating polynomial. The number of past setpoints values used in the extrapolation results from a trade-off between the following considerations :

- The noise in the setpoint is quite important (σ = 25 mrad). Considering a great past horizon will have a filtering and delay effect.
- 10 % of the setpoints correspond to very high evolutive targets. If the filtering effect is important, the extrapolation will be bad and will lead to the loss of the tracked target.

All the MBPC techniques use an internal and linear model of the process. His goal is to predict the process behaviour in the future Unfortunately, the model and the process cannot be identical. The model only approximates the process. Its determination results from a global identification step based on datas meseared on the real process to control. The chosen models for this application are :

$$H_{BM}(s) = \frac{1}{s\,(1 + 0.03s)} \qquad \text{for bearing axis}$$

$$H_{EM}(s) = \frac{1}{s.(1 + 0.024s)} \qquad \text{for elevation axis}$$

As the process and the model are not identical, the process output prediction for the future instants will be biased. The process and model mismatch is known in the past and at the current instant. It can be then extrapolated in the future, that is the purpose of the self-compensator which parameters to tune are :

- degree of the extrapolating polynomial
- number of past values used in the extrapolation
- time response of the process-model mismatch filter

The filter is necessary to smooth the extrapolated datas because of the noise they carry with them.

For the tuning of these parameters, the CAD tool gives an on-line help similar to the above one. Once given the degree of the extrapolating polynomial, the user gets a table in which are given the gain margin, the delay margin and the time response for several values of the filter time response and the number of past values used in the extrapolation. A particular tuning can then easily be chosen. An example of this on-line help is given below (figure 12) for a first order extrapolating polynomial.

*No	nb.dom	tps.filtre	m gain	m retard	tps. rep.
* 1	2	0 100	3 0	0.0200	0 120 *
* 2	2	0 200	3.8	0.0200	0.120 *
* 3	2	0.500	4 8	0 0400	0.120 *
* 4	2	1 000	5.2	0.0400	0 120 *
* 5	5	0.100	5.0	0 0200	0 120 *
* 6	5	0.200	5 4	0 0200	0 120 *
* 7	5	0 500	5 6	0.0400	0.120 *
* 8	5	1 000	5 8	0.0400	0 120 *
* 9	10	0.100	5 8	0 0400	0 120 *
*10	10	0.200	5 8	0 0400	0 120 *
*11	10	0 500	5 8	0.0600	0 120 *
*12	10	1 000	5 8	0 0600	0 120 *
*13	20	0.100	5 8	0 0800	0.120 *
*14	20	0 200	5 8	0.0800	0 120 *
*15	20	0 500	5 8	0 0600	0.120 *
*16	20	1.000	5.8	0.0600	0 120 *

FIGURE 12 : Example of self-compensator on-line help results

To tune all these parameters using the CAD, several steps are necessary.

- Process and model are identical, control variable is not constrained. Reference trajectory, base functions, coincidence points and setpoint extrapolation must be chosen.
- Process and model remain identical, control variable is constrained. Reference trajectory is then adjusted according to the deviation to compensate.
- Process and model are different, control variable is constrained Choose the self-compensator parameters.

During all the procedure, the CAD tool can generate the behaviour of the different interesting datas showing them on time depending graphics :

- Setpoint
- Control variable
- Process output
  Model output
- Difference between setpoint and process output
- Difference between process and model outputs
- ...

For the tracking turret application, the selected parameters are :

- First order trajectory : $t_r = f(c, s_p)$
- Two base functions : step and ramp
- Two coincidence points : t_1 = 0.04 s t_2 = 0.08 s
- Second order setpoint extrapolator with 5 past values

- First order self-compensator extrapolator with 10 past values
- Self-compensator time response filter = 0.5 s

The control law, written in ADA, is the following one.

$$u(n) = k_0.(c(n), s_p(n)).[c_0(n) - s_p(n)] + k_1.[c_1(n) - d_1(n)] + k_2.c_2(n) + v_x^T.X_M(n)$$

Where :

n is the current instant
c is the setpoint
s_p is the process output
$c_j(n)$ are the setpoint polynomial extrapolator constants such as $c(n+1) = \sum_{j=0}^{d} c_j(n).i^j$

$d_1(n)$ is the first order constant of the difference between process and model outputs polynomial extrapolator
$X_M(n)$ is the current model state vector such as $X_M(n) = F.X_M(n-1) + G.u(n-1)$
k_j off-line constants given by the CAD tool
v_x off-line vector given by the CAD tool

The figure 13 shows P.F C. law implemented in the on-board system real time computer Results are shown in the next section.

In appendix B is shown the listing generated by the CAD tool.

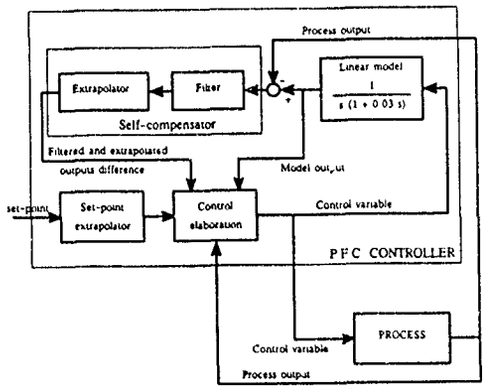


FIGURE 13 : P.F.C. law

3.2.3. RESULTS

The obtained results for a tracking sequence are presented here. The figure 14 shows a step response of the turret and a non moving target tracking. The tracking error remains in the specified values. The rallying time (without overshoot) is very good.

The figure15 shows the tracking of a rectilinear uniform flight at a constant altitude. Here, the target is more evolutive, but the tracking error always remains in acceptable values.

4. CONCLUSION

In this paper, it has been discussed of a new guidance and control technique : Predictive Functional Control. It was shown that this technique is fully compatible with CAD integrated design and allows rapid control laws prototyping. Two applications were presented. Obtained results are quite significant. For the missile guidance law, the miss distance was reduced of 30 to 60 %. For the turret control, it was shown that good performance and robustness could be obtained under wide operating conditions. Compared to the previous SBC Control law, the robustness and the closed loop dynamic have been increased by 50 % and 20 % respectively.

REFERENCES

[1] RICHALET J.: Model BAsed Predictive Control in the context of integrated design. CIM - Europe Workshop on Computer Integrated Design of Controlled Industrial Systems (1990)

[2] H.L. PASTRICK, S.M. SELTZER, M.E. WARREN, Guidance laws for short-range tactical missiles, Journal guidance and control, Vol. 4, N° 2, March-April 1981, pp 98 - 108.

[3] J.L. DURIEUX, Terminal control for command to line of sight guided missile, AGARD, LS-135, 1984, pp 4-1/4-13.

[4] J.IN-JCONG, H. JONG-SUNG, K. MYOUNGSAM, S. TAEK-LYUL, Performance analysis of PNG laws for randomly manoeuvring targets, IEEE Transactions on aerospace and electronics systems, Vol 26, N° 5, September 1990, pp 713 - 721.

[5] DE KEYSER R.M.C. : Model Based Predictive Control Toolbox CIM_Europe Workshop on Computer Integrated Design of Controlled Industrial Systems (1990).

[6] RICHALET J., ABU EL ATA-DOSS S., ARBER Ch., KUNTZE H.B., JACUBASCH A., SCHILL W. Predictive Functional Control. Application to fast and accurate robots. 10th IFAC World Congress, Munich.

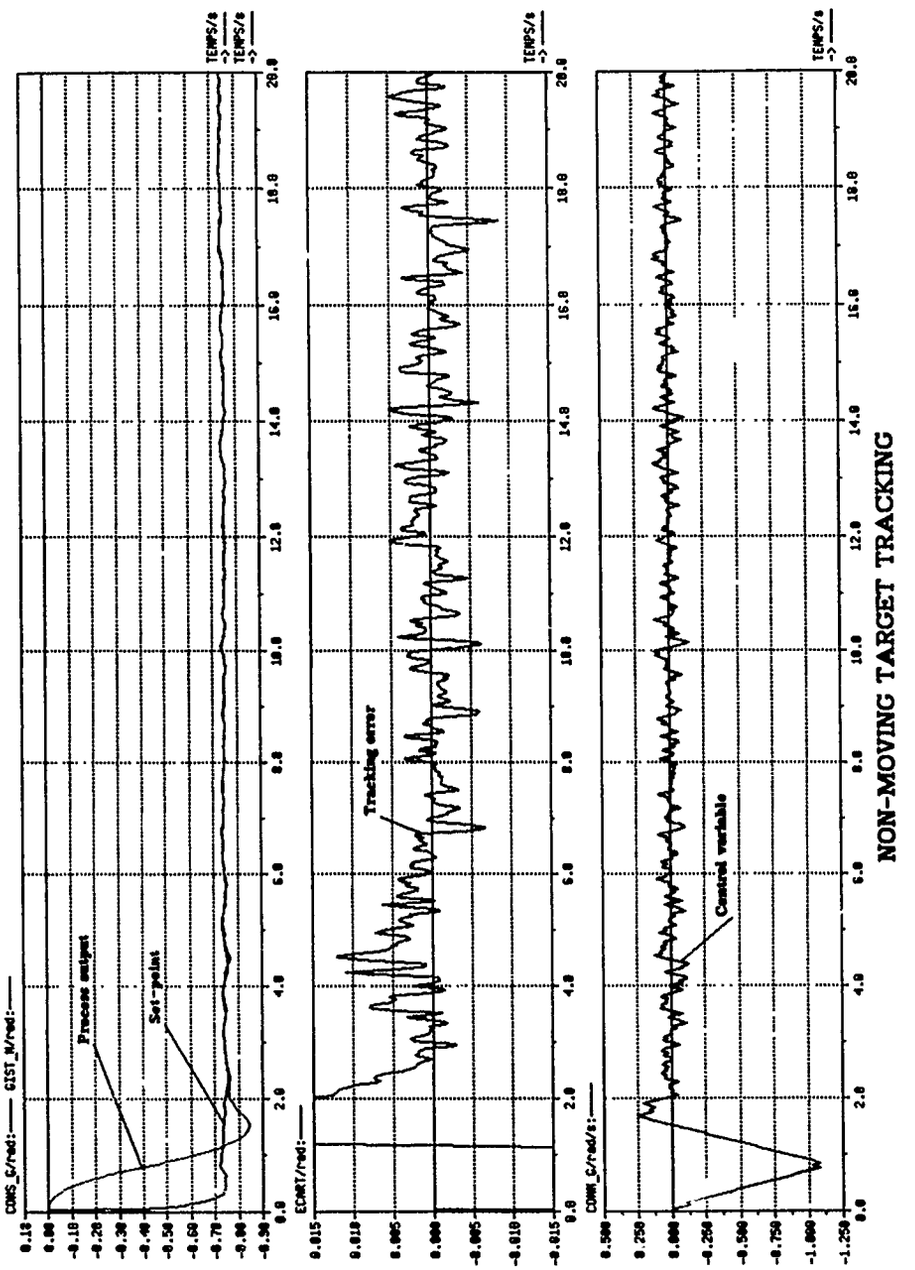


FIGURE 14 : Non-moving target tracking

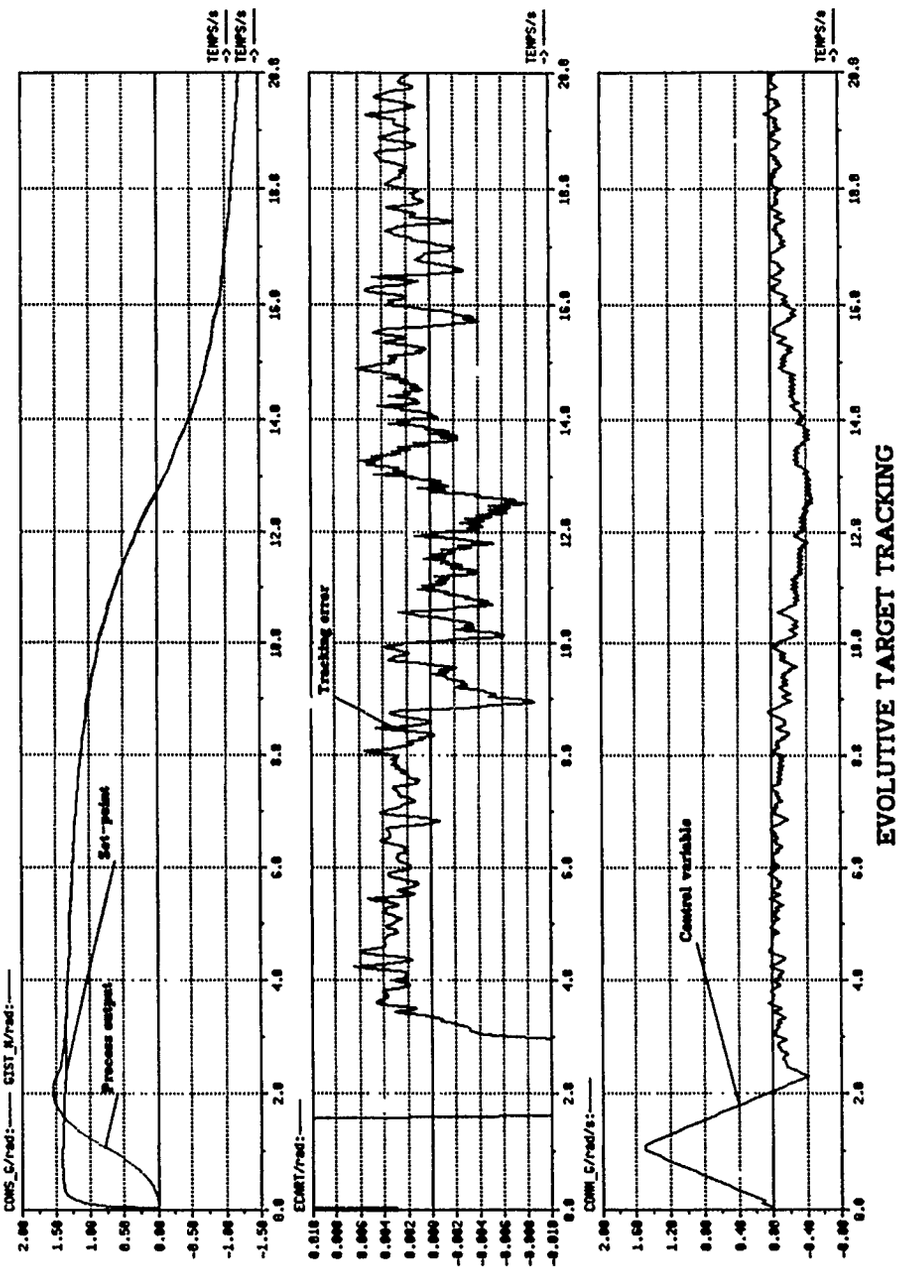


FIGURE 15 : Evolutive target tracking

APPENDIX A

A.1. ON-LINE CONTROL COMPUTATION

The linear model of the process, in stage space form, is introduced in the control elements :

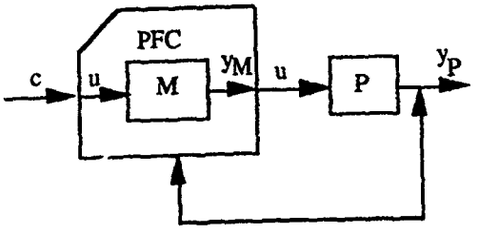


FIGURE 16

with the notations :

P : process

C : setpoint

u : control variable

M : model

y : output

The reference trajectory y_R on the prediction horizon of length h is a first order trajectory defined by :

$$c(n+i) - y_r(n+i) = \alpha^i \{ c(n) - y_p(n) \} \qquad 0 \le i \le h \qquad (A.1)$$

with $0 \le \alpha \le 1$

The future setpoint (known or extrapolated) is expressed in a polynomial form :

$$c(n+i) = \sum_{j=0}^{n_c} c_j(n).i^j \qquad 0 \le i \le h \qquad (A.2)$$

The future control sequence is structured as a linear combination of base functions :

$$u(n+i) = \sum_{k=1}^{n_b} \mu_k(n).u_{bk}(i) \qquad i \ge 0 \qquad (A.3)$$

At each instant, the control calculation is thus reduced to the determination of the coefficients $\{ \mu_k(n) \}_{k=1,n_b}$.

The predicted process output is given by :

$$\hat{y}_p(n+i) = y_m(n+i) + \hat{e}(n+i) \qquad 0 \le i \le h \qquad (A.4)$$

$\hat{e}(n+i)$ denoting the predicted difference between the process and model outputs, obtained through observations on a past horizon, by a polynomial extrapolation :

$$\hat{e}(n+i) = e(n) + \sum_{j=1}^{n_e} e_j(n).i^j \qquad 0 \le i \le h \qquad (A.5)$$

e(n) being the difference at instant n. The procedure of predicting this difference with $n_e \ge 1$, i.e. by considering $\hat{e}(n+i) \ne e(n)$, is called self-compensation.

The control is calculated by minimization of the criterion :

$$D(n) = \sum_{j=1}^{n_h} \left\{ \hat{y}_p(n+h_j) - y_r(n+h_j) \right\}^2 \qquad (A.6)$$

$\{h_j\}_{j=1,n_h}$ being the coincidence points with $h_{n_h} = h$. The number of coincidence points must be at least equal to the number of base functions.

The PFC algorithm yields, for the first value of the future control sequence which is the value to be applied at instant n, the following linear control expression :

$$u(n) = k_0.\{ c_0(n) - y_p(n) \} + \sum_{j=1}^{\max(n_c,n_e)} k_j.\{ c_j(n) - e_j(n) \} + \underline{y}^T.X_m \qquad (A.7)$$

Where the coefficients k_0, k_j and $\underline{y}$ are calculated off-line, X_m denoting the model state vector.

A.2. EXTENSIONS TO THE BASIC ALGORITHM

Constraints on u, $\dot{u}$ and $\ddot{u}$ can be dealt with by applying a particular strategy without modifying the linear control expression. In this strategy, the part of the future model output $y_m(n+i)$ depending on the past is calculated by using the applied (i.e. constrained) control values.

Modification of the criterion (A.6) by adding a quadratic term in u or its variations can be used for control energy reduction and smoother control. The modified criterion is of the form :

$$D(n) = \sum_{j=1}^{n_h} \left\{ \hat{y}_p(n+h_j) - y_r(n+h_j) \right\}^2 + \lambda \left\{ \Delta^k u(n) \right\}^2 \qquad (A.8)$$

This yields to the linear control expression :

$$u(n) = k_0.\{ c_0(n) - y_p(n) \} + \sum_{j=1}^{\max(n_c,n_e)} k_j.\{ c_j(n) - e_j(n) \} + \underline{y}^T.X_m + \beta.u(n-1) \qquad (A.9)$$

where k_0, k_j's, $\underline{y}$ and β are calculated off-line.

Feedforward compensation of a measured perturbation δ is achieved by including a prediction of this perturbation in the process output prediction ; in this case, PFC works with an additional model M_δ corresponding to the $d \rightarrow y_p$ transfer as shown in the figure below :

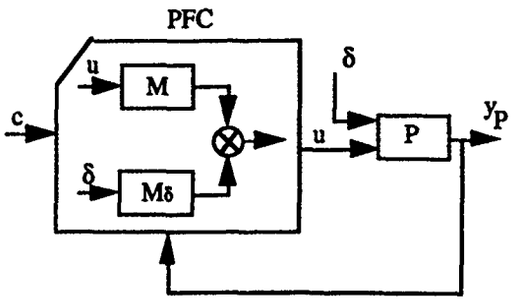


FIGURE 17

There also, a linear regulator is obtained, its expression is given by :

$$u(n) = k_0.\{c_0(n) - y_p(n)\} + \sum_{j=1}^{\max(n_c,n_e)} k_j.\{c_j(n) - e_j(n)\} + \underline{y}^T \underline{X}_m - \sum_{j=0}^{n\delta} k\delta_j.\delta_j(n) + \underline{y}_\delta^T \underline{X}_{m\delta} \qquad (A.10)$$

APPENDIX B

Example of CAD tool outputs for bearing axis.

```
*****************************
PARAMETRES DE LA SIMULATION :
*****************************

periode de simulation =  0.20000E-01 sec.

PROCESS :
---------
     retard = 0.00000     secondes
     FONCTION DE TRANSFERT CONTINUE :
     numerateur   =    1.0000     * p** 0
     denominateur =   0.00000     * p** 0
                  +   1.0000     * p** 1
                  +  0.30000E-01 * p** 2

PERTURBATION D'ETAT :
---------------------
perturbation en entree = 0.00000     +  0.00000     * t
bruit sur sortie : ecart-type = 0.00000     f.coupure = 0.00000     hz

CONTRAINTES :
-------------
     commande max. =   1.5000
     commande min. =  -1.5000
     gradient max. =   1.6000

CONSIGNE :
----------
consigne inconnue generee sous la forme :
consigne =   1.5700     +  0.00000    *t +  0.00000    *t2 +  0.00000    *t3

***************************
PARAMETRES DE LA COMMANDE :
***************************

periode de commande =  0.20000E-01 sec.

MODELE :
--------
     retard = 0.00000     secondes
     FONCTION DE TRANSFERT CONTINUE :
     numerateur   =    1.0000     * p** 0
     denominateur =   0.00000     * p** 0
                  +   1.0000     * p** 1
                  +  0.30000E-01 * p** 2

     MODELE DISCRETISE a : 0.20000E-01 s
matrice d'evolution F :
  1.00000       0.145975E-01
 0.000000       0.513417
matrice de commande G :
 0.540251E-02
 0.486583
matrice d'observation C :
```

```
  1.00000      0.000000

BASE , REF. , COINCIDENCE :
---------------------------
nombre de fonctions de base = 2
     echelon
     rampe
traj.de ref. du ler ordre : tps rep =  0.10000     sec.
nbre de pts de coincidence = 2 :  0.40000E-01  0.80000E-01

EXTRAPOLATEUR DE CONSIGNE :
---------------------------
degre de l'extrapolateur de consigne  =  2
nombre de consignes passees utilisees =  5

AUTO-COMPENSATEUR :
-------------------
degre du polynome extrapolateur de d.o.m. = 1
nombre de d.o.m. passees considerees      =10
tps rep filtre de d.o.m. = 0.50000     sec.

EQUATIONS DU REGULATEUR :
-------------------------
u(n) = k0 [ c0(n) - sp(n) ]
     + k1 [ c1(n) - d1(n) ] + k2 [ c2(n) - d2(n) ] + k3 [ c3(n) - d3(n) ]
     + vx.xm(n)
       -- --
k0 =  47.1561     k1 =  124.841     k2 =  191.746     k3 = 0.000000
vx = 0.000000      -1.49681

EXTRAPOLATEUR DE CONSIGNE :
---------------------------
cm = moyenne des c(n-i) pour i = 0 , ... , horizon de consignes passees
c1(n) = somme sur i de [ yc(i,1) ( c(n-i) - cm ) ]
c2(n) = somme sur i de [ yc(i,2) ( c(n-i) - cm ) ]
c3(n) = somme sur i de [ yc(i,3) ( c(n-i) - cm ) ]
c0(n) = cm - me(1) c1(n) - me(2) c2(n) - me(3) c3(n)
me       =  -2.5000        9.1667       0.00000
yc( 0,.)=  0.58929       0.89286E-01  0.00000
yc( 1,.)= -0.35715E-02 -0.17857E-01  0.00000
yc( 2,.)= -0.32857      -0.71429E-01  0.00000
yc( 3,.)= -0.38571      -0.71429E-01  0.00000
yc( 4,.)= -0.17500      -0.17857E-01  0.00000
yc( 5,.)=  0.30357       0.89286E-01  0.00000

EXTRAPOLATEUR DE D.O.M. :
-------------------------
dm = moyenne des domf(n-i) pour i = 0 , ... , horizon de dom passees
d1(n) = somme sur i de [ yd(i,1) ( domf(n-i) - dm ) ]
d2(n) = somme sur i de [ yd(i,2) ( domf(n-i) - dm ) ]
d3(n) = somme sur i de [ yd(i,3) ( domf(n-i) - dm ) ]
yd( 0,.)=  0.45455E-01  0.00000       0.00000
yd( 1,.)=  0.36364E-01  0.00000       0.00000
yd( 2,.)=  0.27273E-01  0.00000       0.00000
```

```
yd( 3,.)=  0.18182E-01  0.00000      0.00000
yd( 4,.)=  0.90909E-02  0.00000      0.00000
yd( 5,.)=  0.00000      0.00000      0.00000
yd( 6,.)= -0.90909E-02  0.00000      0.00000
yd( 7,.)= -0.18182E-01  0.00000      0.00000
yd( 8,.)= -0.27273E-01  0.00000      0.00000
yd( 9,.)= -0.36364E-01  0.00000      0.00000
yd(10,.)= -0.45455E-01  0.00000      0.00000
```

ANALYST WORKBENCH

Thomas F. Reese, Frank Armogida
Naval Weapons Center
China Lake, CA 93555-6001, USA

SUMMARY

The Analyst Workbench was developed at the Naval Weapons Center to provide analysts with the ability to interactively visualize flight-test and simulation results in the study of missile performance and effectiveness. This technology integrates tools and utilities into one software package that not only assists analysts in gathering data, but that provides the means to analyze and assimilate the data. Visualization technologies--such as the Analyst Workbench--enhance communication between computer and analyst, analyst and analyst, and analyst and management. Current methods are an inefficient use of analysts' time and talents. The Analyst Workbench helps eliminate the large portion of time analysts now spend just searching for the data so this valuable time may be spent analyzing these data instead. Using these technologies to increase personnel productivity and organization communications will ensure the reliability and effectiveness of current and future guidance and control systems throughout the North Atlantic Treaty Organization (NATO).

INTRODUCTION

Analysts involved in the study of missile performance and effectiveness require telemetry data and simulations to conduct significant analyses. As a result, these analysts are inundated with the data generated. Using an exclusively numerical format, the analysts cannot effectively interpret these valuable data.

Analysts need an alternative to numbers. They need the ability to visualize these data and the tools to answer key questions, such as

> Did the subsystems function satisfactorily?
> Did the subsystems function at the proper time?
> Did any evidence of unexpected or marginal subsystem performance exist?

The Analyst Workbench, developed at the Naval Weapons Center, provides analysts with the ability to interactively visualize flight-test and simulation results. The Analyst Workbench supplies the tools needed to answer the key questions listed above. These abilities are imperative to ascertain the integrity of the analyses, to provide insights into subsystem performance, and to share those insights with others.

BACKGROUND

Missile system analysis has been conducted traditionally by means of strip charts and computer printouts. Analysts manually evaluate the strip charts for anomalies for each data parameter on the checklist and write the results on a chart for data-entry personnel to place into the database. After completing the evaluation procedure for each item on the checklist, the analyst conducts a statistical analysis to detect trends within these data. After all the analyses are complete, a report is generated and delivered to the appropriate program office.

Current methods are an inefficient use of the analyst's time and talents. The strip charts and data tapes are never easily accessible. The analyst spends a large portion of time just searching for the data rather than interpreting them. The Analyst Workbench provides the tools required to integrate and make the data accessible so the analyst can conduct a complete and full analysis within a reasonable time.

The interactivity of the Analyst Workbench provides a natural means for an analyst to communicate with data by manipulating their visual representation. This method enables the analyst to control the analysis and find anomalies. Simply, the Analyst Workbench increases the analyst's productivity.

OVERVIEW

The Analyst Workbench is a series of interactive utilities integrated to form a software package. The software components communicate through shared memory and data files. Fig. 1 shows the software components and their stage of development.

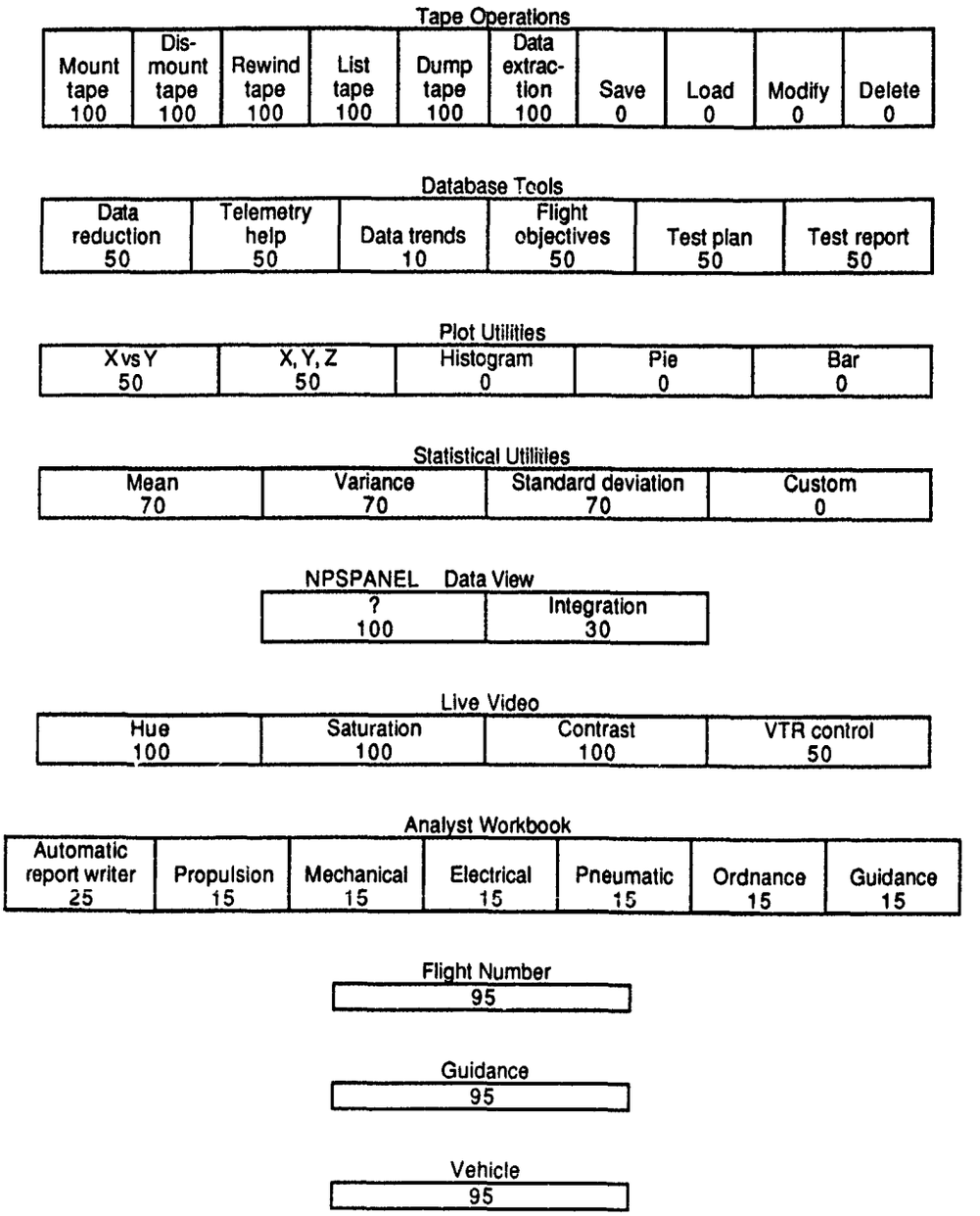


Fig. 1. Analyst Workbench Software Components and Percent Completed.

DATA EXTRACTION AND REDUCTION UTILITIES

Before any analyses can be conducted, the analyst needs to acquire telemetry data. Flight-test telemetry data are normally stored on magnetic tape by test-range personnel. The data tapes are then transferred to the analyst for analyses. The data extraction and reduction utilities select individual telemetry channels from data tapes and install them into the database. For some flight tests, the amount of data can reach up into the gigabytes. Dealing with this "big data" by conventional means requires laborious tape transfers and time-consuming storage-management techniques. Currently, we only have hard disk drives available for data storage. Future plans involve transferring the data to a combination, erasable optical, and worm (write once, read many) optical disk. This storage strategy will provide for on-line access of telemetry and simulation data for the analyst.

The data-extraction utilities are menu driven and usually must be customized for each type of telemetry data set (Fig. 2). Once a format is determined, a computer scientist can write a routine to extract the data and add them to the utility.

TAPE OPERATIONS	PLOT OPERATIONS	UTILITY OPERATIONS	REDUCTION
Mount tape	XY plot	Sort utility	Frequency
Dismount tape	XYZ plot	System utility	Redundancy
Tape format	Pie chart	Convert utility	Manual
Unload tape	Histogram	Make flight path	Automatic
Unload grp	Bar plot	Smooth	Exit
Unload channel	Print plot	Exit	
Exit	Exit		

Fig. 2. Data Extraction and Reduction.

Currently, the reduction utility provides various tools to reduce the amount of data stored. First, the utility evaluates each telemetry channel's update frequency and places the beginning time and update frequency in the header of the file. This procedure eliminates the need to store the time for each time step.

Second, the reduction utility eliminates data duplication. The utility determines if the telemetry value is equal to the previous value, and if so, does not store these data on disk. When the data value does change, the time and value are stored in the database. This simple process can reduce a file size significantly.

Several data-smoothing algorithms were considered; however, in some cases the algorithms eliminated key data elements required for analyses. These algorithms are still implemented, but very rarely are they used. Analysts can usually tell noise from real data.

DATABASE UTILITIES

The database utilities provide a series of interactive information storage and retrieval routines that enable the analyst to query the database for a group of channels, an individual channel, documentation, and overall data trends.

Currently, the database consists of FLTNO, GUIDANCE, and VEHICLE parameters. Each of these icons is shown in (Fig. 3).

When the analyst needs data from a particular flight test, the mouse is positioned above the FLTNO icon and the left mouse is pressed (Fig. 4). At this point, the utility searches the database for available flight-test data sets. The analyst then reviews the list and selects the appropriate data set. This process sets the "current path" in the database and all data are retrieved from this path. To alter the path, the analyst can reselect the FLTNO icon and repeat the process.

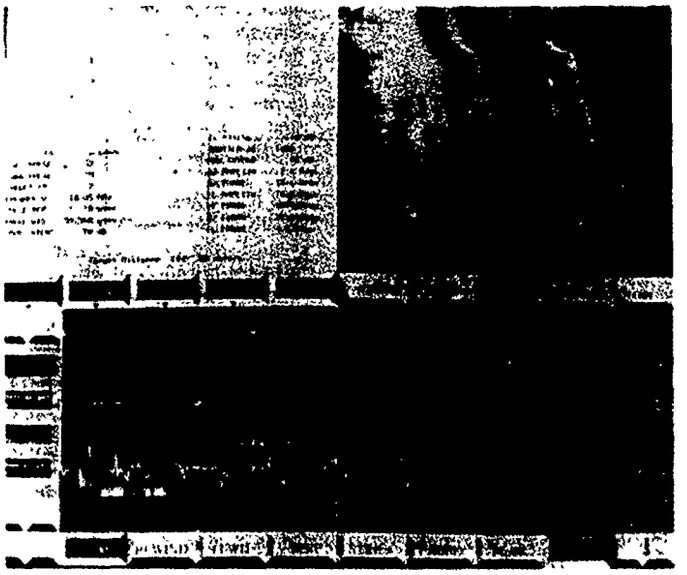


Fig. 3. Data Extraction and Reduction Utilities.

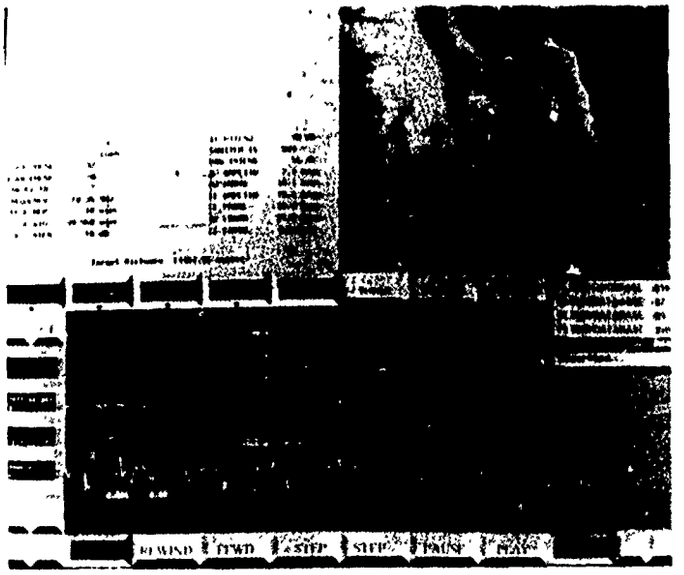


Fig. 4. User Selects FLTNO Icon.

Once the current path is set, the analyst can select the GUIDANCE or VEHICLE icons. The GUIDANCE icon accesses the telemetry data related to the guidance systems, and the VEHICLE icon accesses the telemetry data associated with the propulsion, mechanical, electrical, pneumatics, fuel, and ordnance subsystems. Additional icons can be added to suit the analyst. Once selected, the utility searches the database for available groups of telemetry data and provides a popup menu of the groups. The analyst can move the mouse over the selected group, roll over the menu to get an additional menu of the telemetry channels in that group (Fig. 5). The analyst may now select a telemetry channel that assigns that channel to the "current file." The current file is used by many of the routines described in this paper.

Each of the telemetry channels has an associated documentation file. This file is a text file that discusses the particulars of each channel. The analyst may access these files by selecting a channel by the previous method, then selecting the TOOLS icon, and making a select ion from the documentation menu. The documentation utility then searches for information on the current file and opens a window that displays it to the user (Fig. 6).

Stored in a separate part of the database are the data trends for each of the analysis criteria, such as "fuel burn rate." To review the data trends, the analyst selects the TOOLS icon and then selects TREND ANALYSIS from the menu. The utility then searches the database for a list of available trend studies and presents them to the analyst. At this point, the analyst selects a data-trend study and a window is opened showing a plot of the current trend and the equation used to calculate this value. In addition, the analyst is provided with an additional menu to add, delete, or modify the data in the database.

To conduct a complete analyses, an analyst needs statistical tools to evaluate telemetry data. Currently, the statistical tools available, although rather limited, do provide the basic statistical utilities like mean, variance, and standard deviation.

A tool is currently under development to enable the analyst to rapidly customize his or her own statistical routines and add them to a popup menu (Fig. 7).

The aforementioned utilities and tools provide the analyst with access to all the telemetry data to evaluate flight-test and simulation results. However, analysts need to see the data to conduct their analyses.

PLOT AND TIME UTILITIES

Analysts like to use strip charts and numbers. However, spreading a strip chart over a conference table is not the most efficient means of evaluating the data. The plot and time utilities of the Analyst Workbench provide an electronic strip chart of telemetry channels and a digital readout of the numbers.

The electronic strip chart depicted in Fig. 8 is used to display a time segment of four selected telemetry channels. Using the video tape recorder (VTR) controls, the analyst can move time by pressing REWIND, FFWD, STEP->, <-STEP, PAUSE, or PLAY. This time is stored in shared memory for other processes within the Analyst Workbench to access.

The numerical readouts shown in the upper left corner (Fig. 8) display the telemetry channels' numeric values in the center of the electronic strip chart. The analyst may position the mouse over a readout until a readout lights up and enables the sliding scale for that particular telemetry channel. Once a sliding scale has been enabled, a red light on the lower border of the readout is turned on and the scale associated with that telemetry channel tracks along the mouses horizontal position on the screen. Attached to the left and right sides of the scale's vertical line are rectangles containing the time and value of the telemetry channel. The analyst may move the rectangles by pressing the left mouse button and moving the mouse vertically. To disable the sliding scale, the analyst presses the middle mouse button.

The labels on the left side of the utility display the telemetry channels assigned to each data trace. The analyst may turn a trace on or off by centering the mouse over the rectangle until the rectangle lights up and by then pressing the middle mouse button. To display a different telemetry channel, the analyst retrieves a current file from the database, as mentioned previously, and positions the mouse over the desired trace rectangle until that rectangle is lighted, then presses the left mouse button. At this point the software removes the existing traces data, notifies the analyst of the file being loaded, and then loads the new telemetry channels file into memory (Fig. 9).

The beginning, ending, and current time digital readouts depicted in Fig. 9 enable the analyst to manually alter the time to rapidly increase, decrease the time scale, or move current time to a particular telemetry segment. The ">" and "<" icons are used to move the beginning and ending times incrementally. The analyst positions the mouse over the icon until that icon lights, then presses the right mouse button. The longer the mouse button is pressed, the faster the time scale changes.

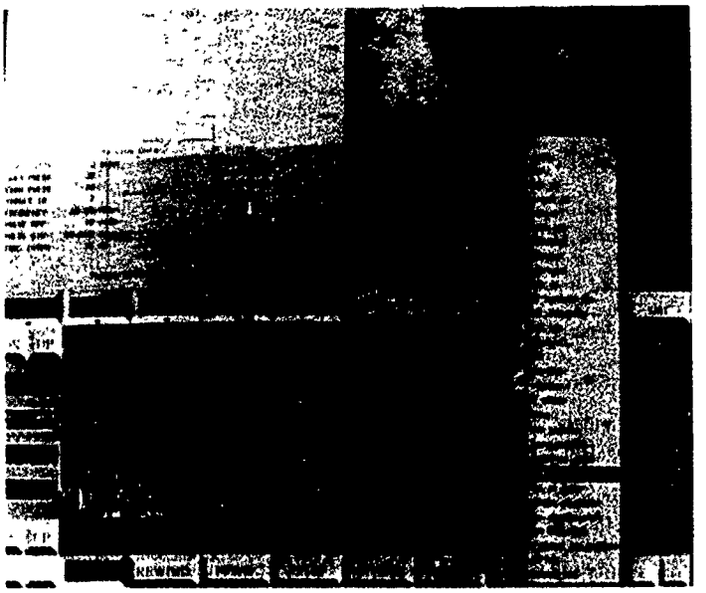


Fig. 5. Guidance Database Menu.

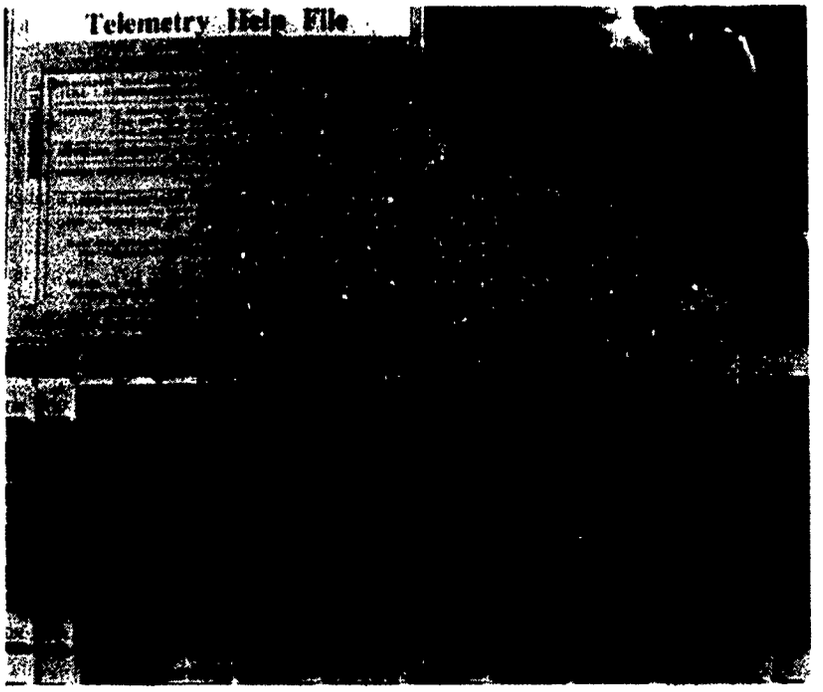


Fig. 6. Documentation on Telemetry Data.

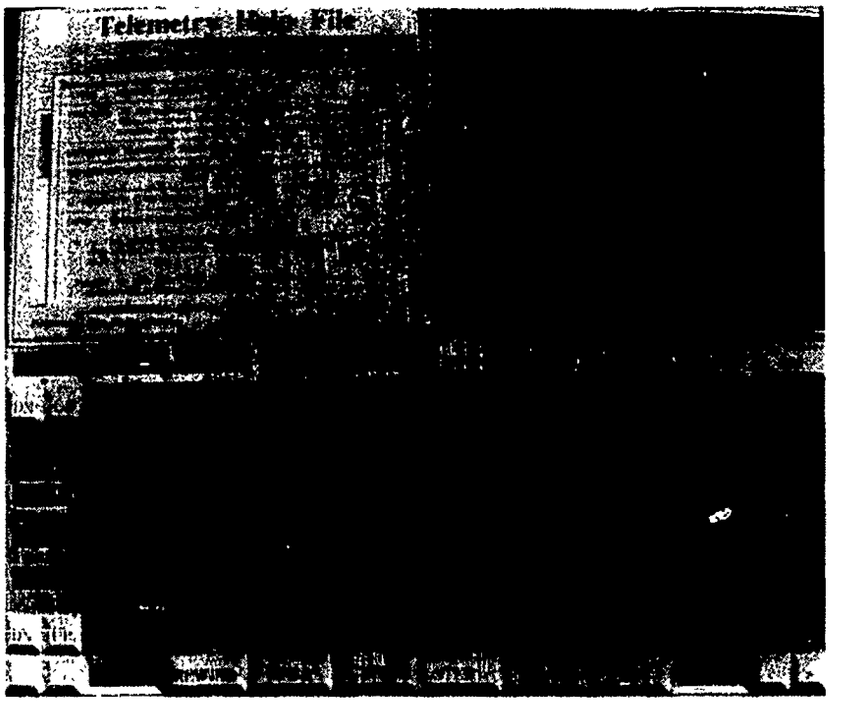


Fig. 7. Statistical Utilities.

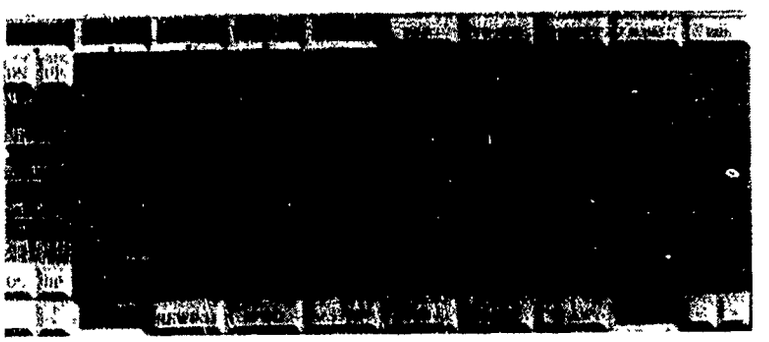


Fig. 8. Electronic Strip Chart.

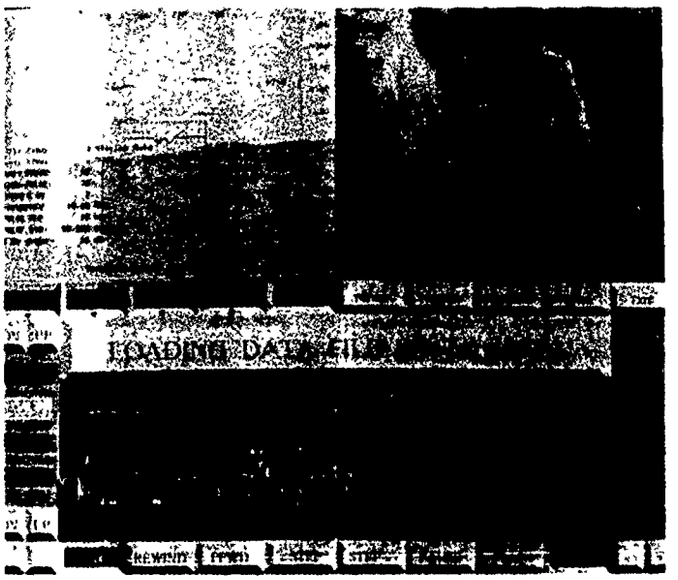


Fig. 9. Loading a New Telemetry Channel.

The vertical scales for each telemetry channel can be modified individually. The UP and DOWN icons shown in Fig. 9 are used to increase or decrease the scales shown on the display. The scales that are turned off are not modified. Again, the longer the mouse is pressed, the faster the scale changes.

The versatility of the plot and time utilities provide many of the visualization tools necessary to analyze telemetry channels. The analyst can rapidly access, display, and manipulate the data used for analysis. The plot and time tool will not eliminate the need for strip charts. Therefore, plans are underway to implement an interface to a strip-chart recorder. However, this utility does satisfy most of the strip-chart needs of the analyst.

OUT-THE-WINDOW TOOL

The out-the-window tool provides a visual representation of the missile flight parameters, seeker characteristics, and physical test environment, Fig. 10.

The U.S. Defense Mapping Agenda (DMA) provides digital terrain elevation data (DTED) and digital feature analysis data (DFAD) to U.S. Government Agencies. This product is widely used throughout the analysis community.

The DTED data are composed of a matrix of elevation data within a 1-degree latitude by a 1-degree longitude quadrangle. The distance between cell elements is approximately 100 meters. Many attempts at generating a realistic out-the-window view for flight simulation have been stopped as a result of the large amount of data. The Analyst Workbench does not attempt to generate a realistic out-the-window view. The out-the-window utility does a simple representation of the terrain and target. The out-the-window scene is a modified product produced at the Naval Postgraduate School, Monterey, Calif. (Reference 1).

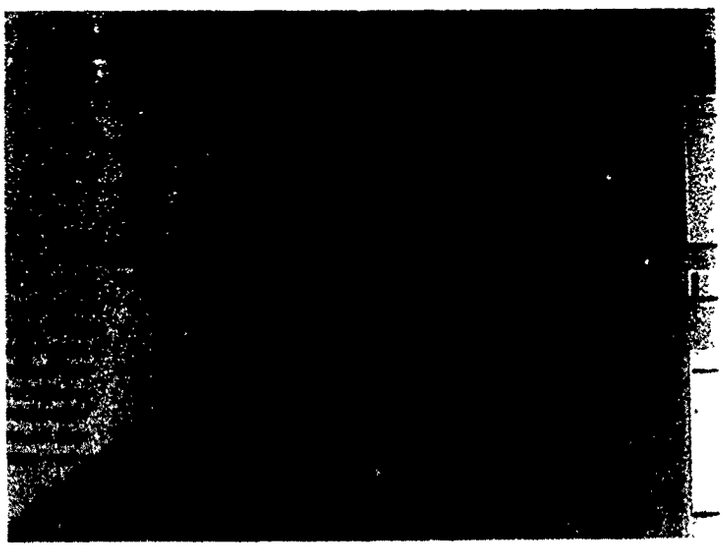


Fig. 10. Out-the-Window View.

The Naval Weapons Center needed to modify this software to fit the needs of the Analyst Workbench. First, the software needed to be driven by an autopilot file and time from shared memory. An autopilot file for each flight test is created from the telemetry and tracking data. The autopilot file is read in and creates a linked list of time, x, y, z, heading, climb, roll, and velocity in internal random-access memory (RAM). Second, the Heads-Up display needs to be modified for each missile type analyzed. This modification includes seeker state variables and seeker field of view. Third, the targets and environment need to be displayed as icons on the terrain.

As the time is modified by the VTR controls, the out-the-window utility alters the perspective of the missile and updates the Heads-Up display. This procedure provides the analyst with visual cues of anomalies within the test. Using the visual representation of the missile-body angles, seeker field of view, and seeker state variables, the analyst can understand and communicate better the complex relationships between the target, environment, and the missile.

When computer graphics, processor, and memory speeds increase, a more realistic out-the-window view will be considered, but now the Naval Postgraduate School's software works fine.

PLAN VIEW UTILITY

The plan view utility provides a three-dimensional perspective of the missile flight on the test range. An icon of the test vehicle depicts the location, altitude, and seeker field-of-view. Icons of the targets and key features are also displayed (Fig. 11).

Using this utility, the analyst can see a perspective view of the missile, seeker range, and field of view. A popup menu is provided, which enables the analyst to zoom and pan into the test area. These features provide the analyst with a better understanding of the seeker interactions with the target, such as several RF-emitter sources.

DATA VIEW UTILITY

The data view utility provides an interactive set of utilities that enables the analyst to tie bar charts, digital readouts, dials, and histograms to an individual or group of telemetry channels. If a data parameter exceeds the maximum or minimum threshold entered by the analyst, the analyst is alerted.

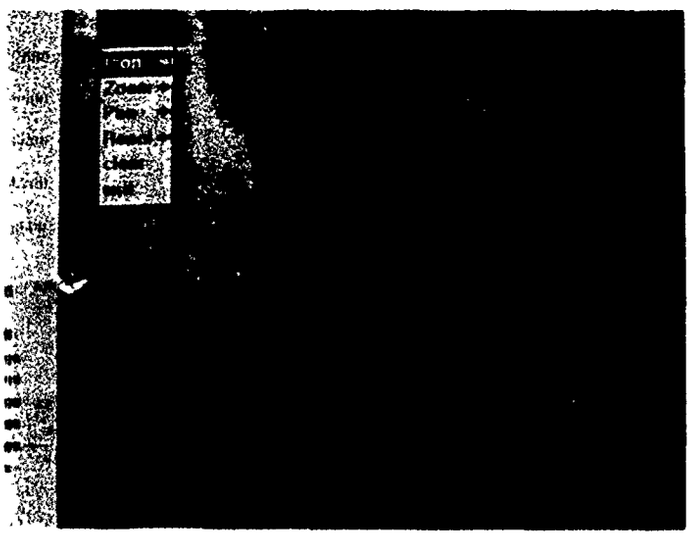


Fig. 11. Plan View Display.

When the analyst selects the data view tool from the TOOLS icon, a window is opened that displays icons for each graph type (Fig. 12). The analyst may then selects a telemetry channel from the current file and assigns it to a particular graph. After the selection is complete, the tool prompts the analyst to enter the minimum and maximum values that are allowed. Then the analyst may place the graph in the desired spot in the window by moving the mouse to the lower left-hand corner and pressing the right mouse button. At this point, the data file is read into a linked list in internal memory, and the display is initialized to the time in shared memory.

As time progresses, the data view tool accesses the time and displays the telemetry value at that time. If the value exceeds the minimum and maximum value input by the analyst the tool alerts the analyst, with a flashing red light and/or a bell sound from the keyboard.

AUTOMATIC REPORT WRITER

The automatic report writer (Fig. 13) is a documentation package customized for an individual missile program office. An analyst interactively accesses standardized forms that guide the analysis and fulfill the laborious documentation requirements of the analyst. After completing the analyses, the analyst saves the completed forms in the database with the data, keeping them together for future reference.

LIVE VIDEO UTILITY

The live video utility provides the ability to display live video in a window on the Workbench (Fig. 14). The VTR-type controls in the plot and time tools advance the tape to synchronize the video with the data. This utility enables the analyst to see and hear the live flight test and help identify critical data segments within the test.

WARHEAD UTILITIES

The warhead utilities provide the ability to study the characteristics of warhead/target interactions. The utilities require input of the missile-trajectory data just before impact, the target name, weapon fuze type, and warhead type. The utilities then smooth the data into a flight path. Then the utility recreates the missile endgame with the weapon/target fuze interaction. The utility displays of the warhead-detonation pattern on the target and calculation of the probability of kill for the endgame geometry and variations.

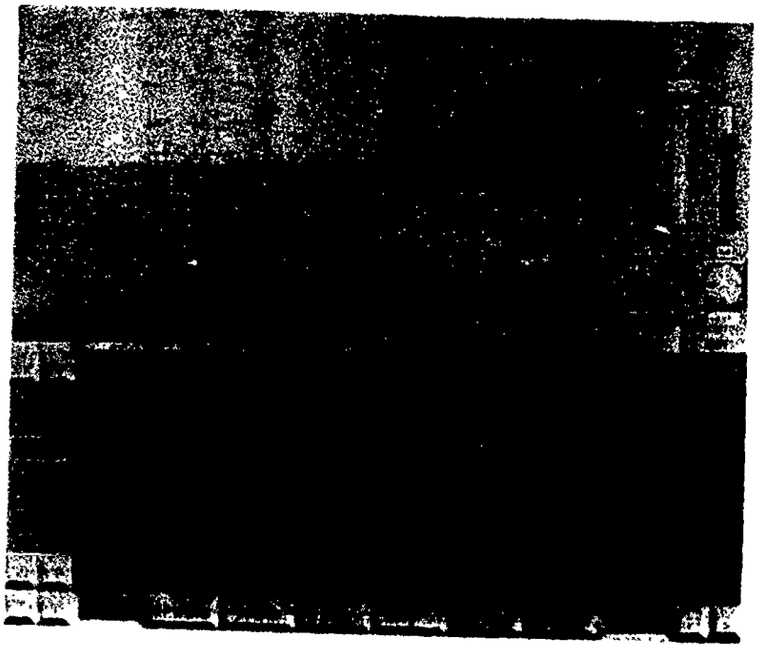


Fig. 12. Data View Graphs.

FUZE SPACE UTILITIES

The fuze space utilities provide the ability to study the general characteristics of various fuze/warhead interactions with a target. These utilities were developed both to understand the interactions and to assist with fuze optimization studies. The utilities display the fuze point, color coded for probability of kill, for a set of parallel trajectories around a target. Multiple sets of data can be overlayed, or plots of probability-of-kill-versus-miss-distance or circular error probable (CEP) can be displayed.

REFERENCES

1. R. B. McGhee, M. J. Zyda, D. B. Smith, and D. G. Streyle, *An Inexpensive Real-Time Interactive Three-Dimensional Flight Simulation System* prepared for USA Combat Developments Experimental Center, Fort Ord, Calif., by the Naval Post Graduate School, Monterey, Calif., 1987, NPS52-87-034.

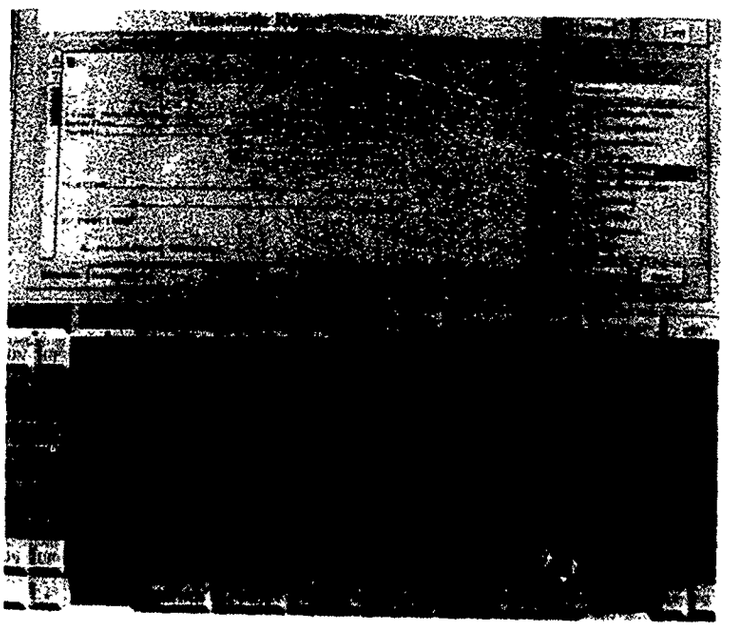


Fig. 13. Automatic Report Writer.

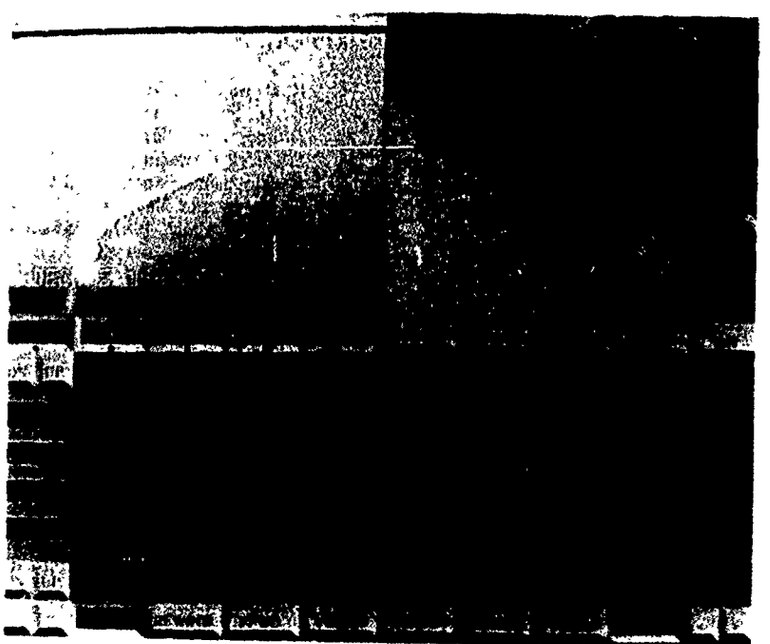


Fig. 14. Live Video Utility.

A PRACTICAL EXPERIENCE OF ADA FOR DEVELOPING EMBEDDED SOFTWARE

by
Christophe GOETHALS - Claude GRANDJEAN
DASSAULT ELECTRONIQUE
55, Quai Marcel DASSAULT
Saint-Cloud
92214
France

Many papers have already been written about the general purpose programming language Ada. The authors of these papers often draw a number of contradictory conclusions such as "users running Ada keep complaining about Ada, but none of them would drop Ada for another language", or "Ada isn't perfect, but it's the best existing language", or "such a language could never be used for embedded applications with stringent real-time constraints".

In this paper we do not claim to draw any final conclusion regarding Ada - it would be too presumptuous on our part to do so in a domain under full expansion - but we do propose some important reflections regarding design methods, real-time aspects, and tools needed, considering our experience with combat aircraft embedded software.

CHARACTERISTICS OF AN EMBEDDED SOFTWARE

Prior to detailing our reflections in subsequent chapters, it is paramount to know the characteristics of the projects on which our experience is based. In fact, the characteristics of a compiler, a configuration manager, or an embedded software are very different.

There are two main types of characteristics :

- Those of the software itself.
- Those of the software development process

Characteristics of the software

Quality Factors

The primary quality factors in the development of combat aircraft embedded software are (if only three are to be retained) :

- Robustness : in fact, mission computer tasks have become more and more complex and even critical regarding pilot safety (in the terrain-following and guidance phases, for example). Thus, the software must be protected against its own defects or an unforeseen behavior of its environment.

- Maintainability and adaptability : an embedded software evolves throughout the life cycle of an aircraft, that is, over roughly 20 years (for example, an average of five modifications per working day during the first eight years of the development process of the MIRAGE 2000 export was experienced). The software's structure as well as the associated documentation are essential elements for the acceptance of such factors.

- Efficiency : in spite of the tremendous improvement in technologies the last few years, it is a must that the memory occupation and computer throughput required by the software be controlled and managed.

Real Time, Response Time, Memory and Throughput Constraints

Because of the nature of the tasks that it has to perform, mission computer software is subjected to such constraints that the choice of the programming language is of utmost importance.

Real Time constraints impose, first, a real-time architecture capable of handling periodic as well as random events and assure a consistency in the processed data set. The choice of what will be called later on the real-time monitor is vital to meet these requirements.

Response time requirements also are fundamental. Indeed, mission computer software controls and manages the information made available to the pilot. To do so, it must assure that the response time between a pilot action, for example, and its acknowledgment on the right-side multifunction display takes less than X ms.

Memory and throughput constraints will be discussed below when the efficiency quality factor is detailed.

Recent Changes Calling For the Use of Ada

Considering the more than 20 year life cycle of a combat aircraft and the improvements made in the computer throughput field, a new quality factor has emerged, and it will have to be added to the previous ones : portability. This is particularly true for RAFALE since the initial operations were carried out on a 68020-based PMF computer system, which will be replaced by a SPARC-based CMF computer system. Thus, the necessity is apparent regarding the choice of language we will have to make to assure portability. Ada was chosen over C because of its higher-level concepts confirmed after a series of tests.

Characteristics of the software development process

Software development process requirements, although they are not significant in the choice of language, are significant in the choice of software development resources associated with this language.

Initial Development Time

The development of software for combat aircraft embedded computers is characterized by a typical incremental development process since the various operational functions constituting the Navigation and Attack System are successively integrated. The development of such an operational function, which can be scaled on the average to ten thousand Ada lines, takes approximately 11 months, as the development process is understood to encompass the following phases : functional definition, global and detailed design, coding, unit tests, integration tests, functional tests, and validation tests. The last two test phases are executed using the target computer itself.

Software Branches and Deliveries

The incremental software development process is concretized by deliveries of the entire software. These deliveries are normally scheduled at the rate of four to six a year. On the MIRAGE 2000 Export we made 44 deliveries in 48 months !!! In addition, these deliveries were not made in a linear manner. Indeed, starting from a common software development trunk we derive what we call software branches which have their own specific lives and are used to debug the operational functions independently of the new developments made on the common trunk. The upgrades made on the software integrated on this branch are integrated in parallel on the common trunk or are only integrated when requested by the client.

Modifications

Because of the complexity of weapons systems and the large number of parties involved (not only government agencies and offices, prime contractors, equipment vendors, but also program managers and pilots), the initial definition of an operational function, and, as a result, everything subsequently related to it, undergoes a vast number of modifications throughout the life cycle, even during the development phase of the function itself. On the average five to seven modifications are made each working day. Here too the methods and the tools must be adapted to these constraints.

SOFTWARE DEVELOPMENT TECHNIQUES

In this chapter we are going to detail the changes that we were led to make in software development techniques to develop embedded software using Ada

These evolutions concern the acceptance of the concept of object, the organization to be installed, and the software programming techniques

Object-oriented development

Ada language offers various possibilities for software quality and productivity enhancements (data typing, packaging, generics,...). But these are useless if an Ada development is not supported by an adequate design method.
For this reason, we decided to study and then use OOD techniques having in mind 2 major advantages they may also offer :

- Reusability because of the better stability of an object over a function, experienced through different projects in the same application domain (aeronautics).
- Prototyping possibilities allowing for early design validation (Ada specifications implementing a design solution can be compiled, due to Ada separated compilation capacity, and even executed if adequate "stubs" are added)

Nevertheless, potential problems had to be solved before full object orientation of our developments.

- System constraints through software requirements documents (our input for software developments) which come from a functional decomposition process and thus, are very likely not to match with object decomposition, through data organization for digital buses exchanges (the way data are gathered into messages and the exchange conditions of these messages often do not correspond to our object decomposition and our update or computation conditions).
- Real-time aspects are most of the time not perfectly handled by OOD methods.
- Configuration management problems may arise from the large amount of Ada units resulting from an object oriented decomposition process.

Organization

To allow for dialog with our client and for technical management of the functions, the software development team assigned to the RAFALE project, which is composed of more than 30 members today, is organized following operational functions criteria.

However, since the concept of object is taken into account in the software's architecture, it is necessary to organize the interaction between the diverses parties acting on these objects. Thus, a team member is assigned to make sure that each object or group of objects is homogeneous in relation to the development of the various operational functions using that object.

In addition, we have defined a dictionary of operational objects, and their refinement at the software level. This dictionary is currently being realized on an object oriented database. It is an indispensable tool for facilitating the distribution of operational knowledge in the objects and the transmission of information to all of the software development teams. In addition, it will be an excellent training tool to discover combat aircraft software from operational point of view as well as from software architecture point of view.

Coding

As we noted at the beginning of this paper, the efficiency quality factor is primary for the type of software which we are interested in. On the other hand, Ada offers a wide range of capabilities in terms of encapsulation, data typing, controls, and so on.

As in other domains, we had to make a trade-off between Ada advantages and efficiency We established Ada programming rules for onboard software.

In particular, three points should be clarified :

- The Ada built-in controls are only used in the unit test and integration test phases, in which we work in native language on a workstation, as we can hardly affort the resulting generated code.
- The exceptions are only used in some very special cases, as the operational software should have a well-defined reaction on a whole host of events, even unexpected ones.
- The generics should only be cautiously used, since the advantage of parameterization provided by them is thwarted in the operational software by the generation of large amount of code.

REAL-TIME STRUCTURE OF EMBEDDED APPLICATIONS

Ada is considered to be weak in the real-time domain, but at the same time Ada is one of the only languages directly integrating real-time features.

When Ada is used for embedded software, and, thus essentially under strong constraints, this problem has to be coped with in a global manner regarding the operational characteristics of the application.

Real-Time Executive

The feasibility of using Ada, including tasking, was one of our major worries. Thus, we developed representative real-time benchmarks of our applications in order to verify their feasibility in Ada. We experimented with diverse types of possible architectures and estal lished the minimal specifications under which the real-time executive we would use, should comply to. These benchmarks permitted us to decide, given a full knowledge of the situation, how to develop mission computer software in full Ada.

The principal characteristic of our embedded applications is the ability to respond very quickly to cyclic external events (a recurrent frequency ranging from 1 Hertz to 100 Hertz) or random external events, along with the additional requirement that all the events are assured to be taken into account. The processings to be executed have diverse priorities and depend on the response time or activation frequency. The internal and external consistency of the data handled and the associated processings are consequently fundamental.

The real-time executive that we developed and are using for our applications has the following characteristics :

- It is compliant to the Ada Programming Language reference manual ANSI/MIL-STD 1815 A.
- It is essentially written in Ada, or in assembly language for those parts linked to hardware or critical in terms of execution time.
- It is not specific to our applications, but is optimized for our specific needs.
- It is configurable according to the target machine to assure the portability of our applications.
- Its underlying real-time kernel includes a scheduler compliant to the Rate Monotonic Scheduling (RMS) principles in order to comply to the hierarchical structuring of the tasks of the applications.

Ada only has a single synchronization/communication mechanism called the rendezvous, which allows a synchronous interaction from n tasks to 1 task.

There is no direct asynchronous mechanism (without blocking the calling task) allowing n tasks to communicate with 1 task or n tasks with n tasks in the Ada programming language. Writing such mechanisms in Ada requires the use of server tasks, thus provoking penalties in memory occupation and execution time.

Our active participation in the ExTRA working group (Extensions for Real-Time Ada) enabled us to be the source of the definition of such mechanisms. The results of this group have been sent to the CIFO/ARTEWG. An effort to obtain a convergence between the ExTRA specification ones is currently under way and is planned for the CIFO 3.0

To satisfy our immediate needs and assure the portability of our applications, we decided to implement the following mechanism :

- The asynchronous rendezvous, thus permitting a calling task not to be blocked (SIGNAL and GO BETWEEN).
- The queueing mechanism (BUFFER).
- The event and pulse mechanism (EVENT and PULSE).

These mechanisms are implementable in Ada, are compliant to the Ada rationale, and assure portability. For our specific needs, we optimized them to cope with the problem of performance by writing some parts in assembly language and by using the internal mechanisms of the real-time kernel.

In order to promote the debugging and validation of software in conjunction with the DEVISOR test tool, the monitor provides :

- A trace of task communications.
- The effective execution time of each of the tasks.
- The storage of the size of the stacks so they can be appropriately sized.

Input/Output

An embedded application as described right from the beginning is strongly serviced by communication systems imposing cyclic or random processings as described above. The number and frequency of I/O are very high. Thus, the communication systems were designed to limit the computational workload induced in the application. To do so, a part of the processings is handled by the coupling boards.

The I/O requests made by the application are :

- Global (several requests are given in a single call).
- Asynchronous (the application does not wait for an answer).

These actions permitted us to reduce the computational workload supported by the application due to I/O facilities by a third.

SOFTWARE DEVELOPMENT RESOURCES

The development of real-time applications under stringent software development constraints such as described above necessitates the availability of an adapted set of software production tools integrated in a true software development system.

Global definition and design tool

STP[1] (Software Through Picture ™) is the specification tool chosen for our onboard software development process. STP is a software analysis and specification environment which is both complete and adaptable. STP allows the combined use of several modeling techniques, which is necessary to specify the multiple aspects of a software system : structured analysis (YOURDON/DEMARCO), structured design (CONSTANTINE), structured real-time analysis (HATLEY), entity-relationship model (CHEN).

In addition, this tool integrates the architectural design phase in which we define our software architecture (both real-time and static architecture). It supports object-oriented design method.

Detailed design, coding and unit testing tools

KEYONE[2] (R) is a tool supporting the detailed design and coding phases (since it has a syntax editor). In addition, it allows one to produce a standardized documentation.

A set of consistent translators (Ada-ALSYS system, assembler) allows one to construct an application from modules written in diverse languages.

DEVISOR[3(R)] is a software debug and test system covering all of the test phases both static (execution not in real-time) and dynamic. Its principal characteristics are :

- The tested software is not disturbed.
- The test is formalized and automated, for easy non-regression testing thanks to a high-level language.
- Independance is assured between languages and target machines.
- The man/machine interface is user friendly and extended use of symbolism is made.
- The test program may be automatically generated.

DEVISOR[3(R)] operates in a static configuration on a host machine (UNIX-based workstation)

Validation Tools

DEVISOR[3(R)] operates in a dynamic configuration connected to a real target machine through a logic analyzer. Thus, it is used for the software integration phases in which the software is integrated in the target computer and the real-time architecture is validated.

The SVB (Software Validation Bench) is used to validate an operational software. It simulates the computer environment (i.e. all the equipments connected to the computer in the embedded system) while complying to real-time and data consistency aspects and allows for setting those states difficult to obtain with real equipment (failures, transmission errors, etc...). The principal functionalities are the simulation of the environment, the graphical display of the behavior of the software to be validated, the control of the target computer, the automation of test procedures for non-regression.

Need for an integrated software development environment

In addition to the tool set that we were forced to develop in part, a true software development environment in which these tools are integrated also was indispensable.

Thus, we developed ILIADE2, which offers the following three functionalities :

- A configuration manager, which controls and manages all of the objects involved in the software development cycle, that is, the documentation (specifications, design, test files, etc...), the code (source, binary, Ada libraries, etc...), the test object (sheets, data, programs, results, etc...) and documents exchanged between a prime contractor and a sub-contractor.
- A production manager, which automates the production not only of the Ada software, but also the documentation, test, and so on. This function also controls and manages the software production servers and their resources in a way transparent to the user.
- A methodological support, which assists the software development teams, project managers, or business executives assuring a presentation and a methodological tracking and follow-up process of the project through an adequate synthesis level.

ILIADE2 is a parameterizable tool, especially regarding methodological support, and an open tool as well, since it allows the integration of new tools.

CONCLUSION : "USING Ada ?"

In conclusion, we are going to conclude about the experience we have acquired using Ada, first in terms of productivity, and then on more general aspects.

Ada and productivity

As we have been developing embedded software, we have experienced a gain in productivity of approximately 30 percent, since the development of a function given equal functionality demands an effort of about 30 percent less.

However, in addition to this gain in productivity, we observed a significant increase in time for the architectural design phase. We also observed the following distribution of the effort needed by the diverse software development phases :

- 10 percent for the functional specification phase.
- 30 percent for the architectural design phase.
- 30 percent for the detailed design, coding, and unit test phases.
- 30 percent for the software integration, functional, and global validation tests.

Ada : success or failure ?

We have seen that the development of embedded software and the embedded software itself are subject to a whole host of constraints and requirements. At the same time the software must be secure and have the lowest failure rate possible. In addition, considering the evolutionary trend in technology, portability is a key factor for long life-cycle software.
All of these factors when combined are favorable or unfavorable for the Ada programming language, which, besides the coding aspects, impact on all of the software development phases.

In light of the experience we have acquired with Ada, it is mandatory that all of the components characterizing embedded software be controlled and managed correctly. That is :

- The software development methods and the way of using Ada to conserve indispensable efficiency.
- The real-time aspects while waiting for future extensions.
- The software development resources to assure productivity.

Only by complying to these conditions will the development of embedded software for combat aircraft written in Ada prove to be successful.

1 STP is a trademark of INTERACTIVE DEVELOPMENT ENVIRONMENTS.
2 KEYONE is a registered trademark of LPS.
3 DEVISOR is a registered trademark of DASSAULT ELECTRONIQUE.

THE DEVELOPMENT OF A REQUIREMENT SPECIFICATION FOR AN EXPERIMENTAL ACTIVE FLIGHT CONTROL SYSTEM FOR A VARIABLE STABILITY HELICOPTER - AN ADA SIMULATION IN JSD.

by
Gareth D Padfield
Stephen P Bowater
Flight Systems Department, Royal Aerospace Establishment, Bedford, MK41 6AE, UK

Roy Bradley
Department of Aerospace Engineering, University of Glasgow, Glasgow, G12 8QQ, UK

Alan Moore
LBMS Plc, 62 Oxford Street, London, W1N 9LF, UK

SUMMARY

In the field of helicopter flight control and handling qualities, the potential benefits offered by Active Control Technology are considerable. To support the development of appropriate handling criteria and carefree manoeuvring features, the UK Royal Aerospace Establishment has been engaged in the development of an ACT system for a research Lynx. As currently envisaged the system includes full authority fly by wire actuation and fail-operate/fail-safe hardware architecture. The impact of the required functionality on the system requirements dictated a need for a precise yet versatile specification of the system, and Jackson System Development (JSD) was selected as a design method since it provides a formal modelling of the pilot interface, and also operates at a sufficient level of detail necessary to ensure completeness and resolution of ambiguities. The tools which support JSD include automatic code generation, and for this work were further developed to accommodate changes to system architecture in an efficient manner. The code produced provides a direct simulation of the design and results in a living specification available for validation and investigation of the written specification.

1. INTRODUCTION

In-flight simulation provides the ultimate validation test of a new flight control concept. The realism of flight test overcomes the deficiences of ground based simulation associated with cue fidelity and modelling inaccuracies. On the other hand, cost and safety issues constrain what is achievable in experimental flight test. A balance between ground and flight test is required to mature a control concept fully. In the field of helicopter flight control and handling qualities, the potential benefits offered by Active Control Technology (ACT) are considerable [1] and results derived from ground and in-flight simulation in Europe and North America have demonstrated benefits at moderate performance levels. Future military rotorcraft will need to operate at considerably higher performance and in tougher environments than currently achievable. To support the development of appropriate handling criteria, carefree manoeuvring features, and the associated technologies in controls and displays, a number of research laboratories are exploring the options for enhanced in-flight facilities. In the UK, at the Royal Aerospace Establishment, attention is focussed on studies into the development of an ACT system for a research Lynx [2,3]. Features of the system as currently envisaged include full authority fly by wire (FBW) actuation, safety pilot with back-driven controls, fail-operate/fail-safe (FOFS) hardware architecture coupled to a range of novel sensors and pilot inceptors providing inputs to the control laws. The FOFS architecture is proposed to enable safe experimental flight in the nap of earth and at the edges of the performance envelope. The impact of this functionality on the system redundancy requirements is considerable. RAE identified a need for a precise, yet versatile, specification of the system to perform these functions developed through design and validated through simulation.

The specification needed to address functionality (for both normal and failed states), operation and performance of the integrated system, together with interfaces, constraints and testing requirements. The specification also needed to be fit for establishing realistic development costs and timescales. The approach taken has crystalised into two phases. Firstly, the development of a textual description of the system with accompanying illustrations. During this activity, a number of different methods were applied by different team members in an attempt to formalise the requirements, to tackle design issues and to provide a format compatible with the later stages of the system life cycle. The Jackson System Development (JSD) methodology was selected for several reasons:

(a) The JSD modelling produces a formal specification of all the pilot/system interaction and so forces the engineer to consider system behaviour from a constructional/design rather than hierarchical description viewpoint.

(b) The JSD network provides a complete description of all the external system interfaces required, plus a systematic partitioning of the system functionality.

(c) Ambiguities in the textual material are naturally identified.

(d) Tools were available to support the method including automatic code generation.

A most important feature of the specification is that it is an executable version of the functional behaviour of the system. Ada code is automatically generated from the specification and, when combined with a simulation of the flight-model and various peripheral devices, becomes a 'living' specification of the system behaviour.

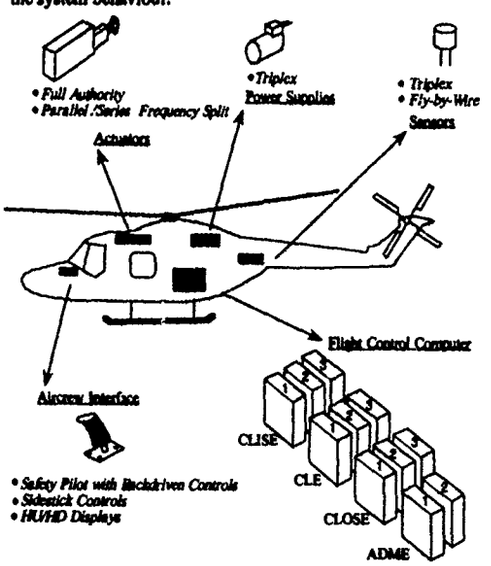


Figure 1 ACT Lynx System Elements

The second class of requirement involves the investigation of options for the exact nature of the final system implementation. This research is intimately connected with the number and types of processing element, and the form of fault monitoring and reporting. To this end a new language has been invented which allows description of hardware layouts and the provision of fault tolerance. The description language is supported by data entry and code generation tools that allow machine manipulation of the

description and realisation of the specified system using Ada. This facility enables the investigation of various hardware architectures, providing the vital realism required to back up more conceptual research.

This paper describes the development of the ACT Lynx requirement specification, drawing on aspects of the system functionality to illustrate the JSD approach of modelling and network analysis. Section 2 covers the evolution of the ACT Lynx requirements leading to the need for a prototype simulation. Sections 3 and 4 deal with the development and use of the Ada simulation, illustrating its investigative potential. Section 5 discusses the way forward for the specification and the project as a whole.

This paper is the second in a series aimed at providing greater visibility on the approach being taken by RAE for the ACT Lynx project. The first covered the life cycle of a control law [4], from conception to flight test, emphasising the validation aspects and proposing a new discipline that enhances the knowledge-capture process while ensuring flight safety. Further papers are planned that will look at the performance versus safety trade-offs in the flight clearance of advanced helicopter flight control laws and software verification aspects prior to implementation in the airborne system.

2. EVOLUTION OF THE SPECIFICATION

2.1 Background

In a series of technical memoranda and reports [2,3,5,6] RAE developed the rationale for a programme of research based on an ACT helicopter. Further studies have demonstrated the practical feasibility of modifying the RAE AH7 Lynx ZD 559 into a full authority ACT vehicle for such a purpose; encouraged by the feasibility of this approach, RAE embarked on the preparation of a specification for the airborne system (Airborne System Requirements Specification) of the ACT Lynx programme. Figure 1 illustrates the design concept where the experimental pilot's conventional control runs are replaced with an ACT system. The elementary modules of the new system are described more fully below, in section 2.3, but, in essence, a flight control computer, with corresponding interface units, connects a new set of inceptors and sensors to a group of parallel actuators driving the original actuation system. From the outset RAE were determined that the specification should be the basis of a well managed procurement exercise, and as such, should solve all of the significant design issues of the system. Potential suppliers would then be able to assess accurately the costs of supplying the various components of the system, since the possibility of being involved in expensive open ended design work would be eliminated. Also, by solving the outstanding design problems *ab initio*, RAE would be sure that the system could actually be supplied in accordance with the specification.

2.2 Adoption of Jackson Techniques

With the objective of producing a complete, unambiguous specification it was decided to employ, as far as possible, the techniques of Software Engineering. The disciplines of these techniques would ensure a rigorous development of the design, and the associated CASE tools would assist in maintaining the precision and integrity of the specification. For the digital part of the total system, the methods could be applied directly but for other parts, which could include analogue, mechanical, hydraulic and even human components, it was not immediately clear how the software techniques could be adapted. Moreover, it was desirable that, at the specification stage, there should be some freedom as to the type of implementation ultimately chosen - leaving open, for example, the option of dissimilar implementations of redundant units. Jackson System Development (JSD) , [7,8] was stipulated as the preferred methodology, but proved, at least initially, to be difficult to embrace in this novel context. While the difficulties relating to the use of JSD were being resolved, there was a partial application of De Marco [9] methods; consequently, when the first version (Issue 2.0 [10]) of the specification was circulated among suppliers, in addition to the conventional structure of paragraphs of text supplemented by a set of technical illustrations and diagrams, it contained a group of data flow diagrams and data compositions relating to a De Marco style enhancement to the text.

As a matter of deliberate policy the design team at RAE took Issue 2.0 and subjected it to careful scrutiny in order to identify those areas where it could be significantly improved. In particular, the possibility of using JSD was re-examined since in the context of the ACT Lynx application a methodology biased towards system development was considered to be more appropriate than a decompositional, hierarchical, descriptive technique. A strength of the Jackson method is that it spans the full range of activity from system definition to production of code [11], so that at one end it is concerned with modelling correctly, for example, the actions of the pilot when he uses the Pilots Control Panel (shown in Fig 2 and discussed fully in section 2.5,below) and at the other end contains the level of

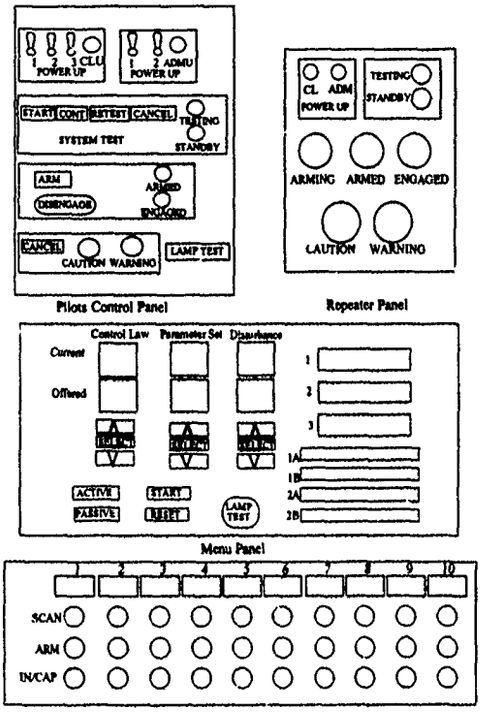


Figure 2 ACT Lynx Control Panels (Schematic)

detailed specification necessary to generate code. Such a level of detail ensures that the design problems of the specification have been addressed even if the software is not actually produced, but in this application a further step has been taken and code produced to implement a simulation of the specified system of Issue 3.0 (section 4). These two areas had not been given sufficient emphasis in the De Marco work, and the discipline of JSD would force attention to them.

2.3 Specification structure.

It was also felt necessary to adopt a modified structure for the specification in order to give it more cohesion. The new structure described the system in terms of its major functional elements. This decomposition was the only one that was imposed on the system *a priori* and reflected a separation which was unavoidably incurred by the nature of the project. Such a subdivision does not preclude further subdivisions should they evolve from the design process. The outcome is shown in Figure 3, where the rectangular components are those relevant to the specification exercise. The bold rectangles are referred to as processing elements and would together form a Flight Control Computers although such terminology was not used in the specification.

The elements of the system are described in the order of the primary flow of the signal information:

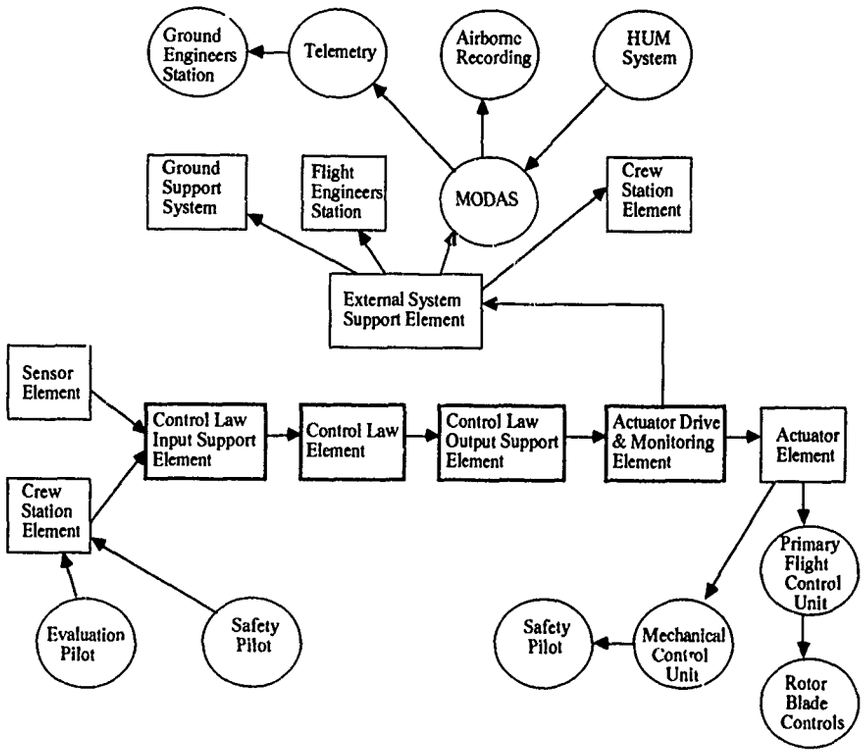


Figure 3 ACT Lynx Logical Elements

(i) Sensor Element (SE). This leading element contains the aircraft motion sensors - attitude and rate gyros and accelerometers, and also the air data units for obtaining velocity, pressure and temperature information.

(ii) Crew Station Element (CSE). The other leading element incorporates the conventional controls for the safety pilot and a versatile side arm controller facility for the experimental or evaluation pilot. The CSE also contains the various interfaces for the pilot to engage, operate and be cued by the ACT system, Figure 2, as follows:

(a) Pilots Control Panel (PCP) - used by the Experimental Pilot for engagement and disengagement and also for conducting the system-test sequence.

(b) Repeater Panel (RP) - provides a copy of the displays for the Safety Pilot.

(c) Menu Panel (MP) - provides other ACT interactions, such as selecting one of the available control laws and sets of parameter values. The same panel provides the interface for injecting preprogrammed disturbances into the system, as part of a flight-test facility used, for example, in the validation of the helicopter mathematical models and in demonstrating compliance with handling qualities requirements of new control laws.

(d) Mode Select Panel (MSP) - available for in-flight selection of control modes.

Clearly the CSE would be expected to feature significantly in any JSD modelling exercise, with the pilot assuming a number of different roles as he interacts with different components of the system. Some of the related modelling issues are discussed in section 2.5, below.

(iii) Control Law Input Support Element (CLISE). The following element has the main purpose of processing and managing the information from the Crew Station and Sensor Elements. It also contains the scheduling of a comprehensive system test.

(iv) Control Law Element (CLE). This next element is supplied with inceptor, sensor, mode selection and related information by the CLISE. The CLE is the raison d'etre of the ACT Lynx since it hosts the experimental control laws which are to be evaluated. It is this element that the user of the ACT Lynx, the handing qualities engineer or flight dynamicist, will interact with. Carefully verified and validated control law software [4] will be plugged into and unplugged from this element. Typically six control laws will be selectable by the experimental pilot with an additional choice of up to six sets of parameters within each law. The demands produced by the CLE for each of the four axes may be separated into low and high frequency demands, if required, which are destined for the parallel and series actuators respectively (An option being currently evaluated). The separation algorithm is part of the user supplied CLE software.

(v) Control Law Output Support Element (CLOSE). The element following the CLE interfaces the demands produced by the Control Law Element to the remainder of the system. It also provides a selectable limiter on the rates and demands produced by the control law as additional protection against immature software.

(vi) Actuator Drive and Monitoring Element (ADME). The final element to provide processing takes the demands from the CLOSE and produces drive signals for the parallel actuators resident in the Actuator Element, and the series actuators in the Primary Flight Control Units (PFCU). The ADME also manages the engagement of the ACT system through the energising of the parallel actuators, and supplies a normal autostabilisation function when the ACT system is not engaged.

(vii) Actuator Element. The parallel actuator system is last in the sequence. When engaged, it drives the existing Lynx PFCUs. The parallel actuators are connected to the conventional control runs from the safety pilot, so that when the actuators are engaged, the controls are back driven to provide the safety pilot with essential control position cues and to aid in recoveries.

(viii) External System Support Element (ESSE). In support of this network of elements is an element which essentially provides a catchment for all of the significant data in the system. It interfaces with the standard data acquisition system MODAS [12] and also with the experimental displays such as helmet mounted or head down displays. A record of all system related events such as engagement, disengagement, and diagnostic messages is retained in a System Journal.

2.4 Element descriptions.

Issue 3.0 of the specification contains a detailed description of each of the elements identified above. As far as possible, the recommendations of the STARTS [13] guide have been followed in the preparation of Issue 3.0 [14]. Each element is described in detail under the headings Type, Function, Operation, Performance, Inputs & Outputs, Interfaces, Testing, and Failure Reporting & Recovery. Where a particular element is composed of replicated units, so that several **units** together comprise an **element**, the replication of units in the element is stated and the unit itself is described under the same headings. For example, the CLISE is a triplex element composed of three identical CLISUs (Control Law Input Support Units). In detail the descriptions are:

TYPE - Some indication is given here of whether implementation is anticipated as an analogue, digital, mechanical, hydraulic, electro-mechanical or human process. The suggested implementation is not intended to exclude alternatives if a supplier possesses a particular specialism The view was taken, after some deliberation, that it was better to make specific recommendations rather than to leave the 'type' issue open. A general allowance could then be allowed for variations that nevertheless complied with the functional aspects of the specification.

FUNCTION - Under this heading is a complete statement of the tasks of the unit , that is,a statement of what the unit has to do. For example, one of the tasks of the CLU (a unit of the CLISE) is inceptor management; the entry reads: "The inceptor displacements and inceptor switch positions shall be processed to provide consolidated signals for the associated Control Law Unit (CLU)"

OPERATION - This sub-section is concerned with how the unit will achieve its functions. This is done by detailed description, in text, of the processing equired for each function. For the CLISE example above, the full details of the processing of the triplex signals would be supplied, including the consolidation algorithms for fault tolerance. The narrative under this heading is used to build the JSD Specification; the full JSD is not held within the text of Issue 3.0, but sufficient initial design work was undertaken to be confident that a JSD specification could be derived from the narrative.

PERFORMANCE - A statement of the times within which the tasks must be completed and, where appropriate, the accuracy that must be achieved. For example, a certain part of the system test must be performed within a stipulated time The sampling rates for the unit would be specified here. A defined constraint is that the total system transport delay should be 25 ms.

INPUTS & OUTPUTS - A list of all signals received by the unit and those transmitted by it. It includes the source of a received signal and the destination of a transmitted one. This information is also presented in diagrammatic form, Figure 4, for example, where the connections to neighbouring units are clearly visible (The network notation is discussed in section 3). There is, of course, a need to maintain consistency here, since for each input listed there must be a corresponding output on some other unit Such consistency is easily maintained by a CASE tool such as Jackson Work Bench [15].

INTERFACES - A list of the units and their types, both internal and external, to which the subject unit is connected. The purpose of this information is to identify the interfacing requirements between units - analogue to digital, for example.

TESTING - A statement of how the function, operation and performance of the unit is verified. In particular this may be done at a system test invoked prior to take off, or by the inbuilt monitoring.

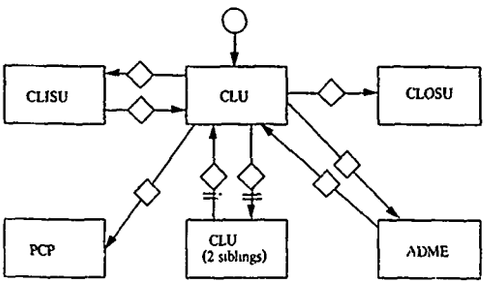


Figure 4 Connections to a CLISU

FAILURE REPORTING AND RECOVERY- A statement of how errors, produced by a fault, having been detected are reported within the system. Usually they are reported to the pilot via the Menu Panel, and they are also sent to the system journal part of the ESSE. Cautions and Warnings may also be raised through the Central Warning System. In addition, a statement of the recovery of the system may be required, often this is by returning to Standby via a controlled disengage - as would be the case when one of the monitoring tolerances within the system has been exceeded.

Once Issue 3.0 of the airborne system specification was complete, it was decided to progress to a full JSD specification in order to check out any residual ambiguity, vagueness or plain error. The full JSD would then be available to use as an adjunct to the written specification. It would give a precise description of the interfaces between the components of the system and between the system and any external devices, to the benefit of prospective suppliers.

A further decision was made to use the JSD to generate a simulation of the ACT system, to produce, in effect, a living specification which could be used to exercise and examine the specification dynamically. The novel features involved in this step are described, in detail in section 3, but the six aims of the simulation in relation to authenticating and potentially enhancing the specification were:

(i) Control and human operation of the system. Pilot evaluation of the procedures for operating the system, for example, the arm/ engage/ disengage sequence can be evaluated through hands on experience. Also suppliers can directly examine the nature of the interface between their equipment and the rest of the system.

(ii) Synchronised control information. The techniques for managing and synchronising control information within an asynchronous system can be verified.

(iii) Establishing tolerances. An asynchronous system generally must allow some tolerance in the monitoring of the information from replicated units. Suitable tolerances can be verified or even deduced.

(iv) Computational load. The processor power and memory requirements of the system can be more confidently deduced from a simulation than a specification. Alternative implementations may be evaluated for processing efficiency.

(v) Fault management. The mechanisms for reconfiguration, and the issuing of caution and warning signals may be verified directly.

(vi) Design Evolution. Alternative designs for the components of the system can be evaluated directly .

Before leaving this discussion on the evolution of the specification in order to consider the development of the simulation in detail, there are two topics worthy of a special note.

2.5 The Supervisor as a modelling issue.

One area which, from the beginning, was subject to intense scrutiny was the control or overall supervision of the ACT system, including engagement and disengagement. Clearly this is a critical area where it is essential to get the specification and implementation correct. An example of an early model is shown as the flow chart in Figure 5, where the System Test, initiated by the pilot, if successful, is followed by a repetition of the arm, engage, disengage sequence of actions. While useful for conveying the general idea of the pilot's interaction in this area, it was not sufficiently precise to base software directly upon.

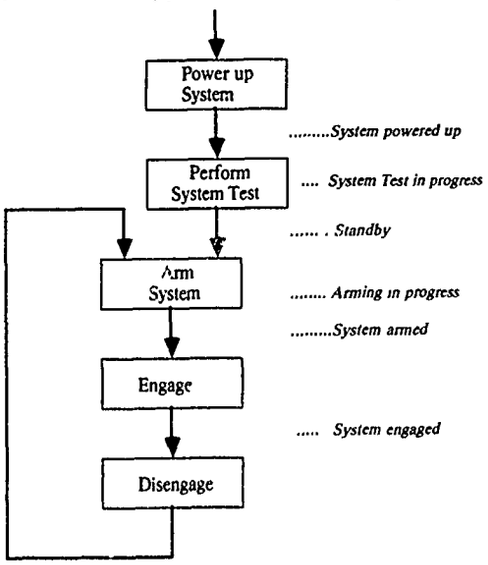


Figure 5 Possible System Control Flowchart

For example, it is possible in the specification, to return to Standby through a disengage action without an engagement of the system. This path is not shown in the flow chart. To express the requirements in a precise manner finite state machines (FSM) were mooted and proved a very useful approach. That shown in Figure 6 included the additional transitions to Standby omitted from the flowchart, but suffered from a shared disadvantage that the system test, itself, included arm, engage, disengage sequences. FSMs have the advantage that they are readily transformed into software so they were seriously considered as a basis for a 'supervisor' process, which would have overall control and only permit allowed transitions of the system to occur. The problems experienced with this approach were twofold. First, incorporating all of the possible

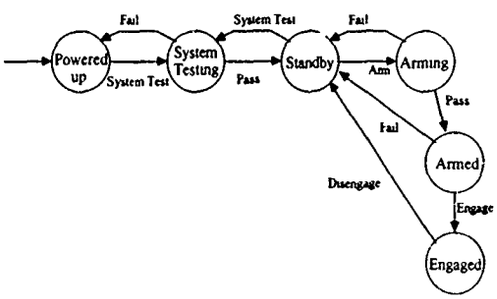
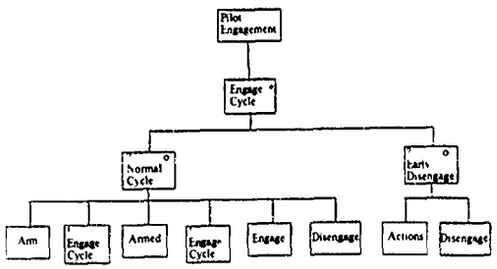


Figure 6 Possible FSM for System Control

states and transitions afforded by the pilot resulted in a very complex FSM, which was difficult to interpret and militated against a correct implementation. Secondly, the engage or disengage actions made within system test, gave different states from those occurring after a successful system test. Consequently, and very importantly, the system test did not exercise that part of the controlling software which would ultimately be used.

These problems were resolved in the final JSD modelling, part of which is shown in structure diagram form in Figure 7 where the system test and engage models are separately treated but have appropriate interlocks. In the structure diagram notation, which is described more fully in section 3, the leaves of the tree structure are actions (similar to transitions of the FSM) and in Figure 7 there is a repetition, denoted by the '*' symbol, of the alternatives, denoted by the 'O' symbol, of a normal engagement cycle or an early disengage. The Arm,Armed, Engage sequence can be quitted, denoted by the '!' symbol, at any stage to continue with an early disengage. The system test process is a simple cycle of alternatives of a successful or unsuccessful test.

The example above has been discussed, for clarity, in a simplified context, omitting such complications as control law selection and disturbance injection, but the same principles apply. The use of JSD in this area helped to achieve a satisfactory modelling and, further, the model can be directly implemented as a process, upon which the whole of the software can begin to be constructed. It is also interesting to note that the separation away from a monolithic supervisor was also guided by the need for maintaining optimum integrity. The various roles of the pilot are modelled separately with appropriate interlocks preventing inappropriate actions, for example, a change of control law when the ACT system is engaged.



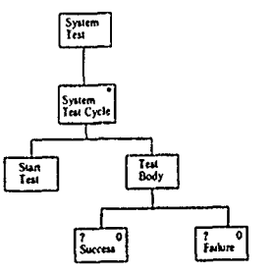


Figure 7 Pilot Engage and System Test Processes

2.6 Fault tolerance and redundancy management

Within the function and operation sections of the unit descriptions consideration must be given to the redundancy managment and fault monitoring issues of the multiplex elements. The main criterion for tolerance is that the system should be first fail operative, and the identification of a fault should alert the pilot to return control to the safety pilot and conventional inceptors, by a controlled disengagement of the ACT System. Faults in a unit are detected by downstream comparison of its outputs with those of its siblings (associated units or lanes within the element - its partners within the redundancy). This recognition is dealt with in three ways:

(i) The consolidation of the redundant signals must not be affected by one signal being in error. There are two type of information to consider here. The first type is 'analogue' or continuous type of data where the median select is used for triplex architectures, the second type is discrete data where a majority vote is employed, both of these are passive fault masking operations used to collect valid data for subsequent processing.

(ii) The error must be recognised and signalled to the system and the pilot via the appropriate panel lamp - this is the monitoring aspect.

(iii)There must be a reconfiguration triggered by the signalling of the error in order to isolate the faulty unit. The isolation is done by ignoring all of the outputs of the faulty unit.

A dual-duplex arrangement operates in a different manner, where each pair of units carries a validity signal and outputs the validity status alongside the functional data. The downstream units can then mask the faulty unit by a tolerant behaviour or reconfiguration. The ACT System has in its initial form, a dual duplex ADME, originally to be compatible with the dual hydraulics of the actuator, and the single fault tolerance arises from the disconnection of a faulty pair of units from the drive to the actuator; the performance of the drive being such that it can tolerate such a reduction of input. The processing elements, CLISE, CLE and CLOSE, are triplex, but have no cross connections at their mutual interfaces .(There is a modicum of sibling monitoring in the consolidation of discrete data.) Consquently they effectively form a single triplex module. The SE and the CSE are essentially triplex with a full number of cross connections to the CLISE.

2.7 The adequacy of the Issue 3 Specification.

The ACT System elements and the functions they performed were conceived and assembled from the combined engineering experience of the project team. This included first hand experience with design of conventional control systems and direct exposure to the helicopter digital flight control system programmes in foreign Industry and Government research laboratories. The FOFS requirement for ACT Lynx combined with the need for significant flexibility in operation created new problems however. The completeness and validity of the upgraded Issue 3 ACT Lynx specification had to be questioned. Were the performance figures achievable in practice? Would there be smooth operation through the PCP? Would the redundancy management logic work? In many projects it is apparent that answers to these kind of questions are deferred until deep into the detached design phase, often when the customer is no longer closely involved. RAE needed to increase confidence and reduce the risk associated with these questions; it was decided to embark on the development of a fully operational prototype simulation. An incremental approach naturally complemented the JSD methodology.

3. DEVELOPMENT OF AN ADA SIMULATION

This section describes the approach taken The organisation of the section is based on the three major parts of the solution, JSD, simulation and code generation. For each there is a description of the approach and a justification for choosing it. Finally, the solution is assessed against the initial requirements for the project.

3.1 An Overview of the Approach

The approach, in brief, was to use the Jackson System Development method (JSD), coupled with automatic code generation in order to produce a simulation of the ACT system. An LBMS CASE tool, Jackson Workbench (JWB) was used to capture the specification of the system, and subsequently to generate the code. The simulation was delivered in six increments, with consultation between RAE and LBMS following each increment.

JSD is used to provide a behavioural description, derived from the textual specification, which is expressed in enough detail to be executable. The whole description is expressed as a network of communicating sequential processes connected via message queues, and with read-only access to each other's data. A separate description is made of the hardware configuration on which this network must run, in terms of hardware units and connections between them. The two descriptions are then mapped one to another by partitioning the processes in the network onto particular hardware units. This mapping is used as the basis for determining the fault detection and recovery configuration. Finally code generation is used to take this complete description and build a simulation which, besides simulating the system, provides a range of facilities such as injecting errors into the simulated hardware and producing diagnostic information for off-line analysis.

The adoption of incremental delivery must be viewed as a success. It was inevitable that RAE when presented with the simulation should find discrepancies between their view of the system and its behaviour, either through misinterpretation of the specification by LBMS, or through inconsistency or ambiguity in the specification. Incremental development and delivery allowed these discrepancies to be identified early and, if desired, corrected.

3.2 Jackson System Development.

Jackson System Development is used in order to analyse the existing textual specification and provide a formal executable specification. The method was jointly developed by Michael Jackson and John Cameron in the early 1980s (references 7 and 8). Since its release it has been used in the development of a number of significant real time systems including:

(a) The control software for a torpedo.

(b) A submarine command and control system.

(c) A wind tunnel control system.

(d) An army wide-area network command and control system.

The technical method consists of three stages, model, network and implementation. These stages are described briefly below illustrated by examples taken from the ACT Lynx project

3.2.1 Modelling. A JSD model is constructed of entities and actions. It is a logical description about a "real world" with which the system must deal. Actions are events which occur in the "real world" which are interesting to the system being built. Entities are the objects in the "real world" which perform. or sometimes suffer, the actions. Figure 8 shows a typical list of actions. Figure 9 describes the order in which a subset of those actions must happen via a time ordering diagram.

Action & Cls	Summary	Attributes
ARM	The pilot requests that the system be armed	
ARMED	The actuator positions and the control law demands are in harmony	
ARM_DEFAULT_MODE	The initial arming of a default control mode	ID_MODE_ID_TYPE
CANCEL_SYSTEM_TEST	A request to cancel the system test	
CAPTURE	This is the signal to mode to go from ARM to ARM_AND_IN_CAP	ID_MODE_ID_TYPE
COMPLETED_SYSTEM_TEST	All tests of the system test have been successfully completed	
CONTINUE_SYSTEM_TEST	Indication that the current test of the system test has been successfully completed	
DISENGAGE	The system has been disengaged This may happen before engagement (1) by the pilot pressing the disengage button or (2) by the system failing to get into the ARMED or ENGAGED state. It may happen whilst ENGAGED on receipt of a signal from an actuator relaying the fact that it has become disengaged	
DOWN_DISTURBANCE_REQUEST	The pilot wishes to be offered the previous valid disturbance, that is the first disturbance with a lower index number (ID) This is equivalent to the pilot presssing the DOWN button	
ENGAGE	The pilot requests (successfully) that the system be engaged	
FAIL_TEST_STAGE	The current 'automatic' stage of the system test has not been successfully completed	
INCEPTOR_VALUE	A new value representing the current position of an inceptor arrives	

Figure 8 Typical List of Actions

Time ordering diagrams are tree diagrams which use JSP notation. The root is named after the entity performing the actions; the leaves (the lowest level boxes which are named rather than numbered) contain the names of the actions performed by this entity. The internal nodes of the tree, i.e. those between the root and the leaves describe different ways of

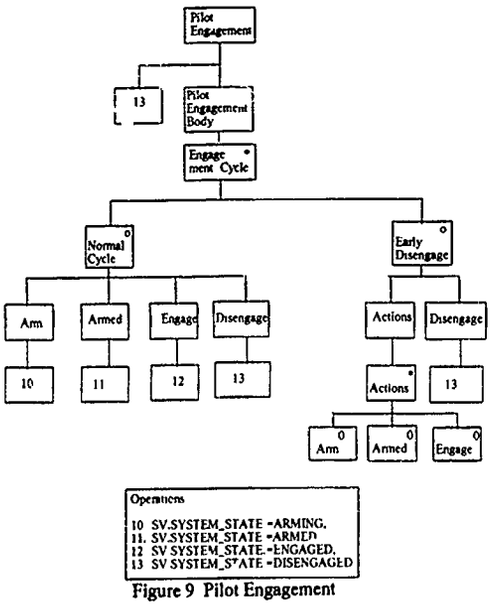


Figure 9 Pilot Engagement

ordering their children and are of three types: sequence, selection and iteration, identified by a symbol in the top right corner of the box. An asterisk (*) represents iteration, that is one or more occurrences; a circle (o) represents selection, i.e mutually exclusive choice, and the absence of a symbol represents sequence, that is all of the children of the node read from left to right. The numbered operations attached to the action leaves indicate how the state of the object (or entity) is changed when it receives the action.

Figure 8 shows a list of actions, that is events of interest to the ACT Lynx system. Some of these actions occur in the time ordering diagram for PILOT_ENGAGEMENT in figure 9; from it, one can see that the standard sequence of events (under the box NORMAL CYCLE) consists of the pilot pressing the ARM button, followed by the system ARMING itself, which is followed by the pilot ENGAGING the system, and finally the system is DISENGAGED. Note that there is an alternative to the NORMAL CYCLE (indicated by the circle in the top right corner); this is called EARLY DISENGAGE and corresponds to the possibility of the system being DISENGAGED at any point. The entire ENGAGEMENT CYCLE can happen many times (indicated by the asterisk in the top right corner).

In a completed model, the total set of tree diagrams describes all of the time orderings of the actions plus the changes in system state which they cause.

3.2.2 Network. In this stage a network of communicating sequential processes is constructed. (Figure 10 provides an example.) The three elements of the notation are processes (boxes), data streams (circles) and state vector inspections (diamonds). The meaning and use of these symbols is described below.

The basis for the network is the set of entities defined during modelling; the state of the entity is recorded in the local data of a process, and the actions become messages passed to the process via an input data stream. Pilot Engagement (The entity described in Figure 9) appears in Figure 10 where its state is inspected by the control law algorithm process. Processes derived from the entities in the model are called model processes. During the network stage new processes are added to take messages from the system boundary and feed them into the model processes, and to use data stored in the model processes in order to generate system outputs.

A JSD process may have many instances, each executing concurrently and each possessing its own local data, collectively known as the state vector of the process. The control flow of all instances of the process is identical, and is described by a tree structure. Processes communicate via message queues (data streams), or by read-only access to each others state vectors (State vector inspections).

The example in Figure 10 shows the connections to the control law algorithm, which inspects data from sensors and inceptors (e.g. the state vector connection CLISE_AMSE_DATA, providing data from the Air Motion Sensing Element), computes new actuator values in the process CONTROL_LAW_ALGORITHM, and then relays them, via the data stream ACTUATOR_DEMANDS, to further processing elements and eventually to the control surfaces.

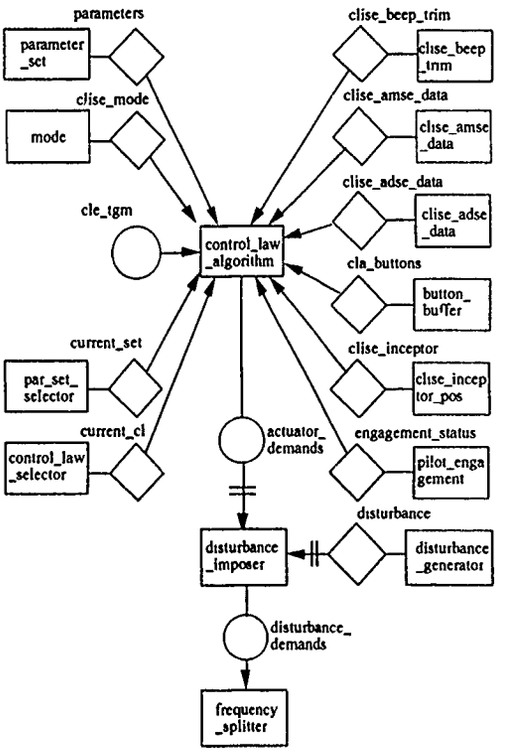


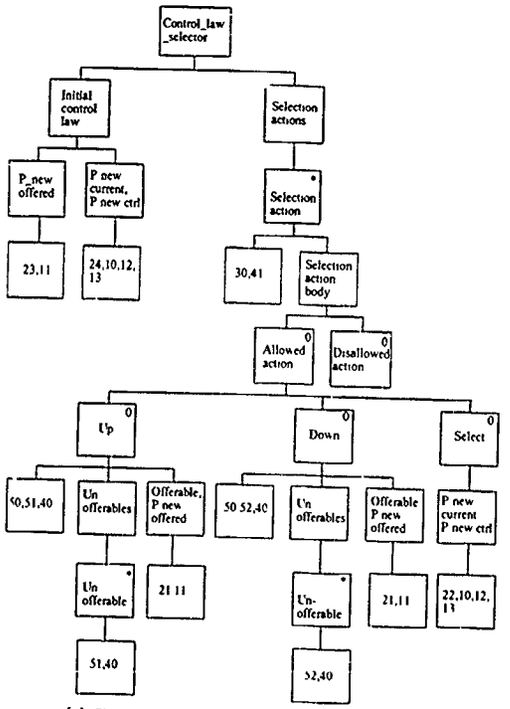
Figure 10 The CLE Control Law

Each process is described in detail using JSP notation; in fact the JSP method can be used to develop the process descriptions. Figure 11 gives as an example the process responsible for interacting with the pilot when he is selecting control laws. The process executes operations which store state as well as reading from input data streams (READ), writing to output data streams (WRITE), and inspecting other processes data (GET_SV). In this way the process is available to gather data and provide responses via its connections.

3.2.3 Implementation. The network which results from the previous step is detailed enough to be executed, but rarely matches the implementation environment: it often has more concurrency than the target (each process instance in the specification executes concurrently with every other), and data stored in the processes sometimes has to be separated out into files and databases.

The implementation step is about fitting the specification to the target environment. It is
discussed in greater detail in the simulation section.

3.2.3 Summary The main aim of the JSD method is to provide a specification which can be usefully viewed from both above and below. The modelling stage is an object oriented analysis of the real world which produces a description which users can readily grasp, because the result is described in terms of objects familiar to to the user. It also provides in an accessible form (tree diagrams) important detail about the model of the real world. The network stage uses two descriptions, one, data flows, which can be presented to the user to indicate the architecture of the system, the other, tree diagrams, which the analyst can use



(a) The CLE Control Law Selector Process

```
10   @WRITE 1 CL_OUTPUTS NEW_CURRENT_CL
              ((ID =>SV.CURRENT_CONTROL_LAW))
11   @WRITE 1 CL_OUTPUTS NEW_OFFERED_CL
              ((ID => SV OFFERED_CONTROL_LAW))
12   @WRITE 1 NEW_CONTROL_LAW NEW_CONTROL_LAW
              ((ID => SV OFFERED_CONTROL_LAW))
13   @WRITE 1 NEW_LAW NEW_CONTROL_LAW
              ((ID => SV OFFERED_CONTROL_LAW))

21   SV OFFERED_CONTROL_LAW = CONTROL_LAW_ID.
22   SV CURRENT_CONTROL_LAW =
              SV OFFERED_CONTROL_LAW.
23   SV OFFERED_CONTROL_LAW = 1.
24   SV CURRENT_CONTROL_LAW = 1.

30   @READ CL_SELECTION_CMDS

40   GET_SV(CONTROL_LAW_ID, PARAMETER_SET_SUBSET).
41   GET_SV(1,CL_PILOT_ENG_SUBSET).

50   CONTROL_LAW_ID := SV OFFERED_CONTROL_LAW;
51   CONTROL_LAW_ID = CYCLIC_SUCC(CONTROL_LAW_ID).
52   CONTROL_LAW_ID = CYCLIC_PRED(CONTROL_LAW_ID).
```

(b) The CLE Control Law Selector Operations

Figure 11

to express the design of a particular function. The resulting specification can be viewed by users from above because it is in terms of their real world and, simultaneously, the specification contains enough detail for the implementers below to perform their task. It is this general property that made JSD particularly attractive for the ACT Lynx specification, although the application was concerned with more than just software. However when a simulation of the specification was envisaged the established features of the JSD method became very relevant. In particular:

(i) Formality and completeness.

(a) Modelling. The semantics of the modelling notation are formal, which allows a formal description of the interaction between the pilot and the system, in terms of the pilot actions and the states into which the system is driven, to be produced from as a result of the modelling stage Figures 8 and 9 provide examples from the project.

(b) Network. The network, once completed, describes the entire functional behaviour of the system to a level of detail which ensures that all functional issues have been aired. In addition, the specification network reaches to the boundary of the system thus providing details of the system interface.

(ii) Fitness for Purpose

(a) Distribution/Concurrency. A completed JSD network is highly distributed, which maps well onto this type of application, where the processing is distributed over many processors.

(b)Separation of Concerns. The specification phase ends with the network still not committed to a particular hardware configuration. Not only does this provide a logical view of the system uncluttered by hardware constraints, but it also allows considerable flexibility when allocating processing to available resources.

(c) Real Time Track Record. JSD has been, and is still being, used on a number of large real-time projects, with Ada as the target language.

(d) Evolutionary Delivery. JSD is a compositional method, sometimes termed "middle-out". In JSD terms, once a model has been built, every new function added provides a potential deliverable, working system. This allows the system to be delivered in a truly incremental fashion.

3.3 Simulation.

Having prepared the JSD design, which in itself has authenticated the textual specification, there is the opportunity to progress to a full simulation by implementing the design This is a definite additional step and it is worth identifying the additional benefits which accrue:

(i) Dynamic and Static Analysis. The essence of building a representative, working simulation of a required system is that it provides unequivocal feedback on the validity and viability of the specification. This feedback is provided via analysis of various forms.

(ii) Verification of Behaviour.The most obvious form of feedback is in the behaviour of the system. Even good textual specifications contain a great deal of ambiguity, and even it the system agrees with the original specification it may not be acceptable. A running system that provides the feed back required to verify the behaviour as described in the original specification.

(iii) Estimation of Hardware Requirements.However good your estimating technique, the more representative data that you can provide to it the better. A working system, even though it may not be entirely representative of the final system provides an excellent basis for estimation.

(iv) Cost and Flexibility (compared to the real system) The obvious choice for an implementation of the specification of the system is to build the system. However in this case that would have been prohibitively expensive. In addition, because of the experimental nature of the hardware configuration, a one off solution did not meet requirements. The obvious choice was simulation.

Given a detailed description of the behaviour of the system using the first two steps of JSD, the final step is to implement the system by fitting the specification network onto the hardware architecture.The ACT system is to be implemented on a multiple node, fault tolerant network of processors with the requirement to perform fault monitoring, fault prevention and fault recovery to provide FOFS system with respect to hardware errors. Both the hardware and helicopter are simulated. The finished system runs on a single IBM PC, or compatible.

3.3.1 Hardware/Infrastructure Description.In order to describe the hardware configuration and the associated fault tolerant infrastructure a new definition language has been created. Descriptions in this language can be entered using JWB and subsequently stored, in the same fashion as the JSD descriptions. Figures 12(a) - (d) provide examples of the descriptions used; Figures 12(a) and 12(b) describe what will be hardware units in the final architecture. A number of options

```
UNIT IE
   STD-INFO
      LONGNAME
      REFERENCE IE
      [*]CLASSIFICATION-SET
      [*]SUMMARY
      This unit is connected to the
      inceptors of the experimental
      pilot.
      [o]NARRATIVE
         NO
MAIN-PART
   [o]TYPE
      ANALOGUE
   [o]BASE-REDUNDANCY
      SIMPLEX
   REPLICATION 3
   [o]UNIT-LVL-SYNCHRONISATION
      ASYNCHRONOUS
         FRAME-LAG
   [*]INTRA-UNIT-CONNECTIONS
UNIT-SID
```

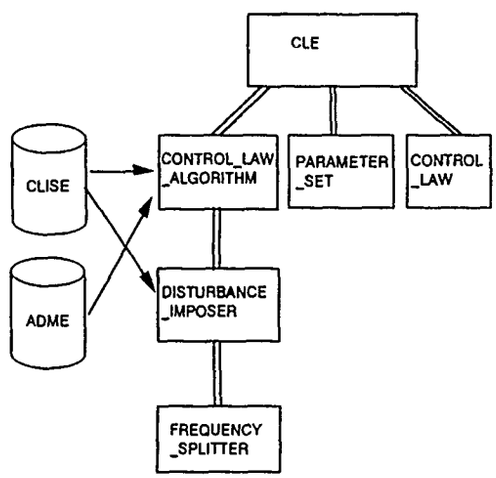
(a) Unit Description

```
UNIT CLE
   STD-INFO
      LONGNAME
      REFERENCE CLE
      [*]CLASSIFICATION-SET
      [*]SUMMARY
      This unit houses the control
      law algorithm and associated
      processing It is the middle processor
      in a three processor "lane".
      [o]NARRATIVE
         NO
   MAIN-PART
      [o]TYPE
         ANALOGUE
      [o]BASE-REDUNDANCY
         SIMPLEX
      REPLICATION 3
      [o]UNIT-LVL-SYNCHRONISATION
         ASYNCHRONOUS
            FRAME-LAG
      [*]INTRA-UNIT-CONNECTIONS
      UNIT-SID
```

(b) Element Description

```
CONNECTION IE_CLISE
   STD-INFO
      LONGNAME
      REFERENCE IECLIS
      [*]CLASSIFICATION-SET
      [*]SUMMARY
      [o]NARRATIVE
         NO
   MAIN-PART
      SOURCE IE
      DESTINATION CLISE
      [o]DATA-TRANSMISSION
         BROADCAST
      [o]SPEC-INTERFACE
         NO
      [o]CONSOLIDATION
         YES
            HISTORY_LENGTH 3
      [o]SIBLING_ERROR_MONITORING
         YES
            HISTORY_LENGTH 3
```

(c) Connection Description



(d) CLE Implementation Diagram

Figure 12.

are provided, including:

(a) The type of unit, analogue or digital (the CLE is one of the main digital units in the system);

(b) How many units (the system is mainly triplex so most of the units have a replication of 3);

(c) Whether the replicated units run in synchrony or not.

Figure 12(c) describes a connection between the CLISE and the Inceptor Element. The options which govern connections are largely concerned with fault tolerance (described below) including:

(a) Whether consolidation is applied to the data, and if so over how many frames the consolidation is performed;

(b) Whether the multiple sources of the data are compared for consistency (downstream monitoring);

(c) Whether the siblings of the current unit are tested for agreement (sibling monitoring);

Figure 12(d) is a pictorial representation of the mapping between the specification network, and a unit (in this case the CLE). The rectangle at the top corresponds to a task type, with the network processes converted via a standard transformation strategy into a procedure calling hierarchy; processes nearer the top call those connected directly beneath them Data external to the CLE is shown via the disk symbols, with access shown.

3.3.2 Implementation in Ada There are many possible mapping schemes between JSD and Ada; References 16 and 17 describe two. The mappings for this project is broadly on that described in Reference 17. This mapping relies very heavily on packages, the aim being to produce a set of Ada packages where each correspond only to one specification object (e.g. process or data stream). This enhances the traceability from the JSD specification to the Ada. The most obvious extensions to this mapping scheme for ACT applications are:

(a) each unit type is mapped onto a task type. Figure 13 shows the Ada for the UNIT described in Figure 12(b). The replication of units is achieved by declaring an array of the task type for the unit with a multiplicity equal to the REPLICATION factor specified in the UNIT object, which in the case of figure 12(a) is 3.

(b) the infrastructure which provides the fault tolerance is described using a number of generic packages which are instantiated based on the information in the connection object.

3.3.3 Implementation of Fault Tolerance. Figures 14(a) and 14(b) describe the connections between the Inceptor Element and the CLISE, and the fault tolerant software sited in the CLISE to handle the data passing between the two units. The fault tolerant strategy was based on that described in Reference 18. The connections between the IE and the CLISE are BROADCAST as indicated in Figure 12(c), that is every IE sends to every CLISE. Figure 12(c) also indicates that each of consolidation, downstream monitoring and sibling monitoring are enabled. The schematic diagram in Figure 14(b) shows the type of fault processing which takes place. Voting is always present where there are many sources for the same data. The voted value is obtained by either majority vote, or median select depending on the type of data. Downstream monitoring implies comparing the values coming from each of the data sources with the voted value. If any of the sources differs for more than a given number of frames (HISTORY-LENGTH in figure 12(c)), then an error is logged and the voter ignores all subsequent input from that source. Consolidation is performed by comparing the historical values from each sibling, gathered over previous frames. A consolidater will only output a new value if it perceives that all of its siblings agree with it. Sibling monitoring implies comparing the voted values coming from the siblings of the unit, rather from upstream sources. Otherwise the processing is identical, with an error being logged when a discrepancy occurs. The sibling which is diagnosed as being in error is then ignored by the consolidation process.

```
with CLE_ID_TYPE_PACKAGE,
use CLE_ID_TYPE_PACKAGE,
with SYSTEM,
package CLE_TASK_TYPE_PACK is
    function CURRENT_ID return CLE_ID_TYPE,
    task type CLE_TASK_TYPE is
        pragma PRIORITY (SYSTEM.PRIORITY'FIRST);
        entry INITIALISE(ID. in CLE_ID_TYPE),
        entry ENSURE_INITIALISATION,
        entry FRAME_START(FRAME_NUMBER in NATURAL),
    end CLE_TASK_TYPE;
end CLE_TASK_TYPE_PACK.
```

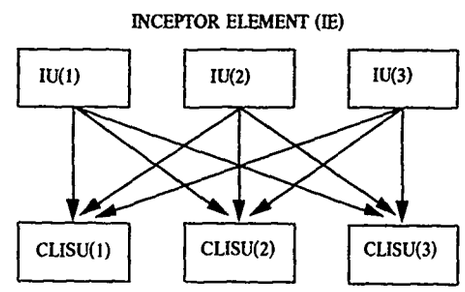
Figure 13 CLE Package Specification

3.3.4 The Choice of Ada as the Implementation Language. The selection of Ada as the implementation language was determined by the following considerations:

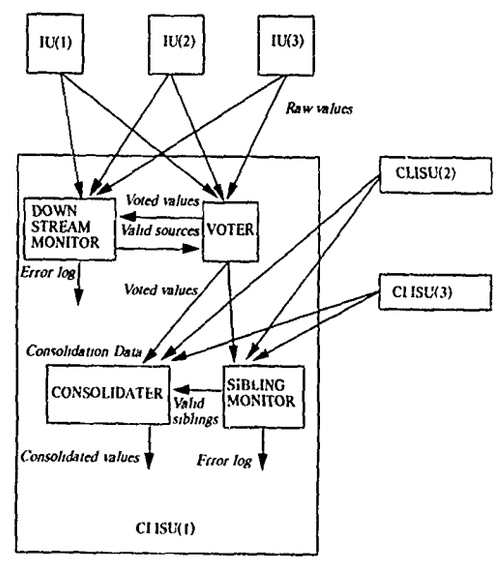
(i) DoD Language. Ada is a DoD mandated language, and is also "highly recommended" by the British MoD, which has provided a large, guarantied market for Ada compilers ensuring a great deal of investment from compiler vendors. This fact coupled with the extensive validation tests required by the DoD has resulted in a number of very high quality compilers.

(ii) Language features. As has been discussed above, packages and tasks have been very important in implementing this system. In addition the comprehensive data typing provided through Ada has enabled a more precise specification to be constructed with a resulting increase in quality.

(iii) Tool Availability. The code generation tool Adacode, described below was already available in prototype form to serve as a basis for the project, and as its name suggests generated Ada.



(a) IE to CLISE interconnection



(b) Schematic Diagram of Fault Processing

Figure 14

3.4 Code Generation.

A significant contribution to the success of the project was the use of code generation. Several factors encourage its use in projects of this nature including:

(i) Productivity. The most obvious gain is productivity. The statistics concerning the number of lines of code and even number of functions (counted using function point analysis FPA)) were very high. The figures obtained from the second delivered increment were as follows:

(a) Function Points per man day 2.34
(b) Source Lines of Code per man day 204

This project could not have been completed within budget and timescales without the use of code generation.

(ii) Ease of Instrumentation. The requirement for dynamic analysis will doubtless change as the simulation is used. Because the system is generated using code generation, the instrumentation can be changed merely by altering the form of the Ada templates and regenerating.

(iii) Evolutionary Delivery. One of the important factors that supports evolutionary delivery is for user feedback at the specification level to be converted efficiently and accurately into implementation changes. With automatic code generation directly from the specification this is assured

(iv) Living Specification. One of the major problems of maintaining systems, especially computer systems, is that the behaviour of the running system diverges very quickly from the original specification of system behaviour once maintenance begins. Code generation provides the ability to maintain a "living specification", i.e. one where changes to the specification are automatically represented in the implemented system.

This feature is especially important in the case of the ACT system because we may wish to evaluate many hardware and error monitoring combinations and even new functions in the course of the planned ACT research.

(v) Re-Implementation. Another important benefit of code generation is that, without changing the JSD specification of the system, a completely different set of code can be generated, for example to fit the system onto real hardware or onto transputers. (This can sometimes be achieved merely by changing the code generation macros, but may require new implementation objects if the implementation is significantly different.) Therefore the investment in a system specification is not compromised when when evolving the system towards greater realism.

Code generation is provided by a prototype Ada code generation tool built by LBMS which has been significantly enhanced during the life of this project. Figure 15 illustrates the workings of the tool. It takes the description of the system, specified using the Jackson Work Bench (JWB) CASE product and generates the complete system from it. Data Extraction is done using built in facilities of the CASE tool. The code templates are combined with the specific parameters extracted from JWB by a proprietary tool called JSP-MACRO. The code generation approach provides a great deal of flexibility with respect to changes in the implementation of the system. Many simple changes can be achieved purely by amending the templates. Even large changes may only require changes to the data extraction, leaving specification of the system unchanged. As well as tools to build the whole system, there are others which rebuild the system regenerating a minimum of Ada based on changes and still others which create test harnesses for any sub network of the specification, providing a cost-effective way of ensuring quality.

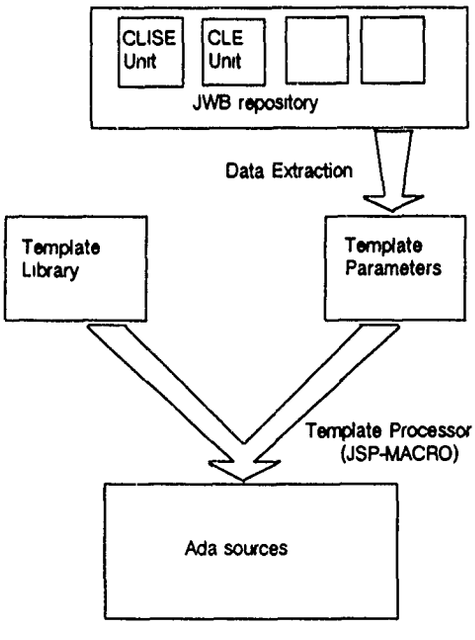


Figure 15 Operation of Code Generation Tool

3.5 Requirements satisfaction.

This section concludes by taking the three major features of the solution in turn, and for each determines which of the initial requirements have been addressed.

(i) JSD:

(a) Resolves Ambiguity - providing feedback which enhances the quality of the original specification.

(b) Formalises Interfaces - so that prospective contractors for specific devices have a precise specification.

(ii) Simulation:

(a) Provides Limited Hands On Experience - allowing pilots to evaluate some aspects of the user interface.

(b) Allows Investigations of Control Mechanisms - to resolve theoretical issues.

(c) Allows Verification of the Acceptability of an Asynchronous Implementation - which is an important and contentious issue.

(d) Allows Investigation of the Tolerance for Critical Functions.

(e) Allows estimates to be made about required processor power and memory usage - via static and dynamic analysis of the system.

(f) Allows verification of the error handling mechanisms - in particular reconfiguration and system test.

(iii) Code Generation:

(a) Allows alternative designs to be verified - including significantly different hardware architectures. This specifically includes a potential switch to a synchronous architecture.

(b) Allows alterations to the testing mechanisms, including monitoring, to be entered and implemented quickly.

4. EXERCISING OF THE SPECIFICATION

With any complex system the problem is always going to be ensuring that the specification is complete in that it totally describes the system behaviour for all eventualities. In addition to complete, it must also be appropriate, correct, testable, unambiguous, and substantiated - together forming the CACTUS rules. This is virtually impossible for any specification written in a non-formal language such as English. The JSD design created from the English specification was to be used to test , in particular, the requirements for completeness and unambiguity, and to clarify the interpretation of the document. The structure of the JSD forces the simulation of the specification to be complete within itself and therefore any omissions in the functional specification must be corrected. Any other inconsistencies become apparent when the simulation of the system is used in a representative way. An example of incompleteness came to light when the first increment of the simulation was exercised. On disengage, the warning light on the pilots control panel was to be illuminated. However, the specification did not include any way to reset (i.e. extinguish the warning light once lit). This omission has now been corrected and, although the example may appear trivial, an important point to note is that even though the functional specification had been read by several people, no-one had detected the absence of such a requirement.

Once the deficiencies in the specification are resolved at the incremental level, the simulation can be completed and exercised. It is in the exercising of the simulation that more potential problems can be highlighted and solved. The first objective of exercising of the system must be to check out the compliance of the simulation with the original functional specification. A thorough check of the simulation is needed to ensure that all the functionality required in a particular increment is present. This is not an insignificant task and careful thought

needs to be given as to how the compliance is to be demonstrated. This task is slightly eased by the incremental approach possible with JSD. The first increment will be fairly simple, hence the correlation between the functional specification and the simulation should be fairly straightforward. As the increments progress, it is only the increased functionality and any associated performance requirements that need to be checked. So not only is the design done incrementally, the compliance checking is done in a similar manner. Each increment demonstrates some aspects of the proposed functionality of the system and can be exercised in a representative way. There may not be same interface as in the cockpit, but the basic pilot / system interaction can be adequately simulated using a keyboard. This ability to interact with the simulation was particularly useful and resulted in some changes to what can be referred to as the Pilot Vehicle Interface (PVI). For example, the proposed system state lights on the pilots control panel were changed because it was felt that the exact status of the system at some points in time was not obvious.

Prototyping systems in order to optimise the PVI is of course not new and several specific prototyping languages have been developed to perform that role. In addition, however, because the design includes the description of the functionality of the system, using the JSD approach offers the opportunity for examining the ability to estimate of processor and memory requirements. Instrumentation of the simulation means that whilst exercising the system estimates of the computational power of the required processors can be obtained. These results can be fed back into the non-functional parts of the of the original specification It has always been a contentious issue in prototyping as to whether the software produced in this phase of the development cycle should be used in the final version of the system. The argument against using prototype software has always been that during its life there will have been frequent changes and patches so that, at the end of the prototyping phase, the software is very unstructured. However, because the changes to the prototype software under JSD are always done a the high level design and not at the code level, the code always stays well structured and suitable for use in the final delivered system.

One aspect of the specification writing which has been particularly difficult to quantify is the specification of tolerances In any redundant system, where the comparisons are made for fault checking purposes, tolerances need to be set against which the system can check itself. If the tolerances are set too close then the system will signal an error where none exists, if the tolerance is to wide then the system may not detect an error where one does exist - or alternatively may not react sufficiently quickly to it. In complex systems, several tolerances need to be set at various strategic points throughout the system. Without putting those tolerances into a representative system and then testing that system it is difficult to ascertain what level they need to be to provide the right level of protection The simulation allows the tolerances to be input and easily changed so that the effect of various levels and combinations of tolerances can be established. The simulation not only allows the examination of tolerance levels with a correctly functioning system, but also allows a representative selection of faults to be injected to establish the behaviour of the system.

In a modular system such as the one being developed here (see Figure 1), it is vital to define rigorously the interfaces between the various elements to enable changes and upgrades to be easily made to the sytem. JSD enables each module to be treated separately and the design method enforces a rigorous approach to the specification of the interfaces. This has the added advantage that different system architectures can be tried for parts of the system and still retain the integrity of the generated executable code. Thus quadruplex systems can be substituted for triplex, dual duplex substituted for triplex and so on. The implication of such changes can be investigated by a further exercising of the system This ability is not only valuable during the initial stages of the development cycle, but also later when actual solutions are being proposed to meet the specification. The simulation can be modified to represent, and then exercised to ensure compliance of, any proposed solution; thus reducing the development risk

In summary, the ability to exercise a simulation of the proposed system confirms that the specification writers ideas and assumptions are actually valid. It is a very valuable tool which allows a rigorous checking of the completeness and consistency of the original specification. The exercising of the simulation ensures the adequacy of the design and allows initial processor performance estimates to be made. By running the simulation of the proposed system, fault tolerances can be set - a task that is virtually impossible to do purely from a theoretical point of view. Alternative architectures can be implemented, and due to the modular nature of the design method, these alternatives can replace the original design and tested for compliance. Thus the tool has a valuable role to play all through the development cycle.

5. THE WAY FORWARD

At the time of writing the ACT Lynx project is at a hiatus. Estimated procurement costs for the system and its certification are high and are likely to require a multi-partner team to be affordable. Both UK and international options are being explored but no clear way forward curently presents itself. Activities in support of the project are continuing at RAE including the study of performance / trade-off issues associated with trials in flight (safety) critical areas. The role of the safety pilot is crucial to this work and ground-based simulations [18] have been conducted - and are planned - to address critical functional questions such as optimum location of disconnects, backdriving frequencies, mismatch tolerances for failure managment, and PCP ergonomics. In parallel with these topics, the requirements specification will continue to be developed. The current operation form is essentially complete in its functionality. Future tasks include:

(a) Instrumentation of the simulation to evaluate end-to-end and internal performance and behaviour.

(b) Production of a comprehensive user guide to the simulation.

(c) Comprehensive exercise of the simulation to validate the specification across the operating spectrum.

(d) Upgrading the requirements specification in line with the results of (a) - (c)

(e) Upgrading the requirements specification to include a second level of JSD analysis, i.e. network and process diagrams together with text.

(f) Implementation of the Ada simulation in real time with representative pilots', engineer's and software development stations

Many of these tasks can be embarked upon concurrently and are not specific to the implementation in the final system. The results from these activities have generic value and can be used to guide and support similar projects for example The current Ada simulation has been developed in seven increments and the approach has demonstrated the utility of this approach. The simulation has 'grown' in a controlled manner with each increment offering more functionality for review and revision if necessary. The approach has had the added advantage of enabling the software engineers to develop their understanding of the application incrementally. A top-down approach to the design would have required considerably greater investment in 'application learning' before any creative work could have been started. It is recognised that there are contentious issues in system development and that there are no right or wrong approaches. JSD has exposed functional anomalies and forced hidden issues into the open through its emphasis on design, however. The behaviour of the ACT Lynx system, as currently configured, is now well understood - the flight critical nature of the application makes this an attractive position to be in.

6. CONCLUDING REMARKS

The handling qualites opportunities offered by active control technology for helicopters require considerable research effort using both ground and in-flight simulation before the final potential is realised Much work has already been done but the peculiar problem areas, such as carefree handling, of high performance levels have yet to be explored in-flight. The safety-critical nature of such flight research demands that a fail-operate

design concept be employed covering both system hardware and software. In the UK, the Royal Aerospace Establishment has proposed the procurement of an experimental ACT system for its research Lynx. This paper describes the development of the requirement specification for the airborne system including crew station, sensors, processing elements, actuation etc. In its current form the requirement is a textual and diagramatic description of the system behaviour covering functionality, operation, performance, testing and interface requirements The specification is supported by design using the JSD methodology. An outcome of the design work is a prototype Ada simulation of the system. Examples of the JSD modelling and the mapping into Ada have been described. Initial results from exercising the simulation have been presented. Although the overall ACT Lynx project is on hold until an affordable package is defined, the requirement specification continues to be evolve, with an upgrading scheduled to follow from a comprehensive instrumentation and exercise of the simulation. A real time implementation is planned which could form the core element of a ground system to support software development.

7. REFERENCES

1 Padfield G. D., (Editor), Helicopter handling qualities and control. Proceedings of the R.Ae.Soc Conference, London, 1988.

2. Winter J. S. & Padfield G. D., A discussion paper on an ACT flight research programme using the RAE Bedford Lynx. RAE Tech Mem FS(B) 523, 1984.

3. Padfield G. D. & Winter J. S., Proposed programme of ACT research on the RAE Bedford Lynx. RAE Tech Mem FS(B) 599, 1985.

4. Tomlinson B. N., Padfield G. D. & Smith P. R , Computer - Aided control law research from concept to flight test. AGARD CP 473, 'Computer Aided System Design and Simulation', 1990.

5. Winter J. S., Padfield G. D. & Buckingham S. L., The evolution of active control systems for helicopters; conceptual simulation to preliminary design. Proceedings of the AGARD FMP Symposium on ACS, Toronto, 1984.

6. Thomson K., The results of the WHL feasibility study in support of the RAE Bedford flight controls research programme. Systems Technology Note STN 19/84. Westland Helicopters, 1984.

7 Jackson M., System Development. Prentice Hall, 1983.

8. Cameron J.R , JSP & JSD: The Jackson approach to system development IEEE Computer Society Press, 1983.

9. De Marco T., Structured analysis and system specification. New York: Yourdon Press, 1978.

10. Wright B.P., RAE ACT Lynx -Airborne system requirement specification, Issue 2. WHL Flight Control Department Note FCDN 88/05, 1988.

11. Birrel N.D. & Ould M.A., A practical handbook for software development. Cambridge University Press, 1985.

12. Jewel C., MODAS analysis system - system overview. Prosig Computer Consultants, 1986.

13. DTI/NCC. STARTS Purchasers' Handbook: "Procuring software-based systems". NCC Publications, Second Edition, 1989.

14. RAE. RAE ACT Lynx Airborne system requirements specificiation Issue 3.A. 1989.

15. LBMS, Jackson Work Bench User Guide. (In preparation, 1991)

16. Cameron J.R., Mapping JSD Specifications into Ada. Proceedings of the 6th Ada (UK) Conference, 1987.

17. Lawton J.R. & France N., The Transformations of JSD Specifications in Ada. Ada User, Jan 1988.

18. Kimberely A. & Charlton M., ACT Lynx Safety Pilot Simulation - Trial Runaway. RAE FM Working Paper (89) 031, June 1989.

8. ACKNOWLEDGEMENTS

The authors would like to acknowledge the contributions of Westland Helicopters Ltd and Theta Analysis and Systems Ltd to the work described in this paper.

Software Methodologies for Safety Critical Systems

by

W.C.Dolman A.M.Ashdown
Lucas Aerospace Limited
York Road
Hall Green
Birmingham
United Kingdom B28 8LN

T.C.Moores
MOD(PE)
St Giles Court
1-13 St Giles High Street
London
United Kingdom WC2 H8LD

SUMMARY

UK MOD(PE) identified Ada[1] as the single preferred high level language for the implementation of defence real-time operational systems from 1 July 1987. This meant that projects selecting an implementation language after that time must select Ada, unless there are sound and documented reasons for using an alternative.

UK (MOD)PE therefore decided to invite proposals for the High Order Language Demonstrator (HOLD) to examine the applicability of Ada to an aero gas turbine FADEC, and awarded the contract to Lucas Aerospace Ltd, Birmingham. This paper describes the work carried out to date by Lucas Aerospace on this contract.

1. INTRODUCTION

UK MOD(PE) identified Ada[1] as the single preferred high level language for the implementation of defence real-time operational systems from 1 July 1987. This meant that projects selecting an implementation language after that time must select Ada, unless there are sound and documented reasons for using an alternative.

The major potential benefit of the application of Ada to military systems is the reduction of Life Cycle Costs (LCCs). In addition Ada is a truly international standard and as a result very wide support, in terms of Programme Support Environments (PSEs) and industry expertise can be expected. Ada also provides facilities for structured design which holds the prospect for a modular approach with verifiable and re-usable software components.

UK MOD(PE)'s concern was that Ada was not yet ready for incorporation into full development of high integrity software based systems, such as flight safety 'critical' Full Authority Digital Engine Control Systems (FADECs).

A FADEC is a real-time control system. The control requires a fast execution time, typically in the order of 20ms, and all of the functions must be computed within this time frame. The main functional activities are:

i) Input handling, including sampling, validation, averaging/filtering and scaling.

ii) Control law computation.

iii) Output handling, possibly including status and fault code data.

iv) Fault monitoring and detection, state input checks and Built In Test (BIT).

v) Fault handling, take action to implement fault procedures, for example change control lane or set the system to a safe state.

There are two main areas of concern:-

Firstly the lack of visibility of the object code and its characteristics.

This is due to the way in which high order language (HOL) sourced object code is generated. Software is written in the HOL, and then converted by an automated process, compiled, into the object code that will be loaded into and used by the target system. Whilst Ada lays down stringent requirements for the design of compilers, and the compilers have to be formally validated, there remains a doubt about their integrity, and certainty of the object code produced actually representing that required by the source, and therefore their suitability for this application.

Secondly the Size of code.

The use of the full Ada language with many compilers was understood to be inefficient in the production of object code, compared to the specialised lower-level languages being used for aero-engine control. This would result in many times more computer memory space and processing power being used for a given function, and would be a serious limitation to the use of Ada for aero-engine control. However, restrictions on the features used, and careful optimisation of the source code might greatly alleviate the problem. It was expected that there would be a speed and power penalty arising from the use of Ada, but it was considered possible that the penalties could be reduced to an

[1]Ada is a registered trademark of the US Government (Ada Joint Program Office)

acceptable level.

UK(MOD)PE therefore decided to invite proposals for the High Order Language Demonstrator (HOLD) to examine the applicability of Ada to an aero gas turbine FADEC, and awarded the contract to Lucas Aerospace Ltd, Birmingham. This paper describes the work carried out to date by Lucas Aerospace on this contract.

2. PURPOSE AND SCOPE

The purpose of the HOLD programme is to examine the applicability of Ada to a military aero gas turbine FADEC. However, it is clear that much more can be undertaken whilst pursuing the top level objective. To ensure that the programme is as "real" as possible the contractor has been required to base the programme on an existing in-service UK military FADEC.

The programme therefore covers:-

i) The identification of those features of the Ada language which conflict with the requirements for a flight safety "critical" aero-engine control system.

ii) The utilisation and critical assessment of design and development methods that will provide the best possible application of the language to this type of system, to meet both performance and integrity requirements.

iii) Re-programming of an existing flight certified Engine Electronic Control (EEC) in Ada.

iv) The assessment of the efficiency of the executable code, and the resulting system performance and integrity, using the existing flight certified EEC as a benchmark.

Within these major activities the HOLD programme will generate much valuable information on topics such as:-

i) Considerations leading to the selection of the compiler.

ii) Considerations leading to the selection of the support environment.

iii) Development of the Ada solution.

iv) Simulator rig testing utilising a reprogrammed EEC.

v) Assessment studies to identify the benefits and penalties of the use of Ada.

vi) Selection of processor/computing power to implement an Ada solution.

3. REQUIREMENTS OF A SAFETY CRITICAL ENGINE CONTROL SYSTEM

This section of the paper attempts to describe the features and life cycle of an Aircraft Engine Control System which may set it apart from other avionic embedded software systems. The main purpose is to highlight the main differences so that some of the decisions described later in the paper can be better understood.

FADEC system software is generally set at the critically level of "Level 1" software as defined by RTCA/DO-178A. The "Radio Technical Commission for Aeronautics, DO-178A Software Considerations in Airborne Systems and Equipment Certification" defines Level 1 software as:-

> Functions for which the occurrence of any failure condition or design error would prevent the continued safe flight and landing of the aircraft.

Although RTCA/DO-178A is a civil certification standard recognised by both the United States of America and European certification authorities, it is the standard which was adopted for the flight certified engine control system being used in HOLD. Consequently, it was also adopted for the HOLD programme so that direct comparisons between the flight certified engine control and HOLD could legitimately be made.

It is worth pointing out at this stage that software, used for engine control, requires a computer system plus the associated input and output conditioning for it to operate and communicate with the outside world. The design of this system is of paramount importance, as the split of functions between hardware and software, with the safety features embeded in both, provides the safety critical system. Software running in isolation does not constitute the total safety critical system, and consequently cannot be considered in isolation.

3.1 Description of a FADEC

3.1.1 FADEC Functionality

A FADEC comprises all the sensors, actuators and computing elements that realise engine management. The (EEC) is a major component of the FADEC. It is beneficial to later sections of this paper to segregate the 'essential' FADEC functions from those functions arising as a result of 'how' the system is implemented.

3.1.2 Essential Functions

The primary purpose of the FADEC is to control a gas turbine installed on an aircraft throughout the flight envelope. Thrust demands from the cockpit and flight management computer are input to the EEC. The EEC utilises a set of control laws and schedules primarily based upon engine and

airframe measurements of pressures, temperatures and speeds to control the prime interfaces to the engine, including fuel metering, variable guide vanes, engine igniters, engine bleed valves and thrust reversers.

3.1.3 Additional Functions

The FADEC design is required to meet stringent integrity and reliability requirements. The architecture of the FADEC and of the EEC is designed to maximise fault accommodation. The additional functionality required of the EEC is:

i) To condition, calibrate, validate and select input signals (critical inputs are normally duplicated).

ii) Validate correct operation of and select output drives.

iii) Dormant fault detection.

iv) Redundancy management.

v) Store all fault data for subsequent retrieval so that the status of the FADEC can be established.

vi) Provide test features to aid unit development.

3.2 Development Life Cycle

One of the major differences between an engine control system and other avionic systems is the software life cycle. It is not unusual for a development life cycle to span 10 years or more and during this time, the software process must be sufficiently flexible to accommodate numerous changes. Some of these changes will be required to be carried out in short time scales, possibly less than 24 hours during particular stages of the development life cycle such as engine test cell operation. The EEC life cycle began in 1979 and is still a live project with software modifications being planned for this year. A typical project for civil application can last as long as 5 years with modifications to the software taking place after entry into passenger service.

The life cycle normally begins with software requirements that are incomplete. This is totally understandable as the life cycle begins while the engine itself is under development. The engine, in return, requires a basic control system to operate it so that the engine can be run and the development process continue.

So we have a situation where both the engine and its associated control system are under development and continuous modification. Thus the software teams do not have the luxury of frozen, complete and unambiguous requirements until quite late in the project life cycle.

The majority of the development changes are implemented in the software system due to the fact that it is easier, but not necessarily cheaper, than modifying the hardware system. Therefore the software environment employed must be flexible but must provide a top quality product without reliance upon formal verification, as this task is impractical during the engine development process.

3.3 Real Time Control System

The term "real time" means many things to many people, from data entry systems, supermarket checkout systems, banking systems to control systems to name but a few. The consequence of failure to carry out a certain task in a certain time for a real-time engine control is an error with a significant consequence, not merely an inconvenience. When this situation arises it must be dealt with in a correct and safe manner, not ignored. This feature of engine control systems has a profound effect on the hardware and software systems which must react quickly and safely to a timing error. The requirements for the engine control system must contain the specific time requirements and any system used to provide and analyse these requirements must have the ability to take time into account.

3.4 Software Verification

The verification of software is a very important, if not the most important, stage of a software life cycle. The outcome of the verification stage is to show that the software meets the requirements and only meets the requirements. There are many facets of verification including functional testing, structural testing, code review, static testing etc. To be able to meet the safety levels required for safety critical software the testing methodology employed must be against the target code resident in its target environment. If testing is carried out against source code, emulators, simulations etc then this will only be acceptable if the following conditions apply:-

i) The source code to target code process has itself been verified or is completely analysed.

ii) The tools used in the process are themselves verified to the same level as the software criticality level.

3.5 LUCOL[1]

The EEC currently in production was programmed using the Lucas Aerospace LUCOL Programming system.

The basis of the system is a high level application oriented language consisting

[1] LUCOL is a trademark of Lucas Industries Plc

of a series of LUCOL Modules representing commonly used analogue control system blocks together with sequential logic operations, input, output and safety routines. The control engineer solves his problem by specifying an appropriately ordered network of LUCOL Modules. These LUCOL Modules are drawn from a library of rigorously tested microprocessor targeted assembler language programs.

Each LUCOL Module has a mnemonic identifer and a standard functional diagram assigned to it. The control engineer draws his system block diagram using the LUCOL Elements - this block diagram then forms a pictorial representation of the software.

The basic control source program is generated simply by producing a calling sequence listing the LUCOL Modules, in mnemonic form, and their associated parameters. These parameters specify:-

i) The data flow between LUCOL Modules (analogous to the signal flow on a conventional block diagram).

ii) The direct parameters such as gains, time constants etc.

A feature of LUCOL is that a LUCOL Module may use the output of the previous LUCOL Module as an input automatically. This method of transference, termed "signal flow" is employed by most LUCOL Modules and results in an improvement in the efficiency and clarity of the resultant control program. However, in a few cases, explicit flow is used, the input or output being defined in the calling sequence. Typically this method is used in such cases as hardware interfacing LUCOL Modules where several parallel operations are likely. This again is to optimise overall efficiency.

4. HOLD METHODOLOGY

The approach adopted for the development of HOLD was to reprogram one lane of an existing FADEC unit. The unit chosen was a dual lane EEC i.e. it had two identical independent lanes of control. Each lane of control was for dry engine control only, there being a third common lane of control for reheat. The advantage of choosing such a unit was that we could replace one lane with the code developed for HOLD whilst retaining the original software of the other lane. This meant that during testing of the unit we could freely change lanes between the two different versions of the software to examine performance.

The starting point for the software development was the software requirements that had been used to develop the original software. We could have started further back along the development path at the system requirements level but we felt that this approach would lead to different design implementations, that would throw into doubt the result of any comparisons made between the two systems. Likewise we could have started further down the development path using the design of the original software as a template for the Ada software. This approach was also rejected. Although it would have given a very good basis for a comparison of the physical effects of the two systems, it did not render sufficient flexibility in the design process to explore all of the inherent structured design features of Ada. So the middle road of choosing the software requirements was chosen as the optimum starting point, bearing in mind that we would have to keep a close eye on the design approach to ensure that the resultant Ada software was consistent functionally with the baseline software implementation, and thus did not invalidate any results of the comparison.

Once this starting point was chosen, the next stage was to decide on the implementation approach to be adopted for the specification and design of the Ada software. As part of the HOLD programme we wished to investigate the impact that Formal Methods would have on engine control systems and to see where such methodologies would yield benefits in terms of improved software production and integrity.

We considered implementing the whole of the software requirements using a Formal Method but decided that this approach was too great a risk to the programme. The time required to perform a full Formal Methods implementation was an unknown as was the effect that it would have on the subsequent production of Ada software. As the other main aim of the programme was to investigate the suitability of Ada in an engine control environment we did not wish to risk an approach that may fail at the first hurdle.

There was also the unknown regarding how well we could implement, using Formal Methods, the present software requirements due to their structure and layout. We had to be sure that any new representation was consistent with the present software requirements and included the same functionality.

The next step was thus the selection of the approach to be taken to represent the software requirements.

4.1 Requirements Capture

We decided that the most advantageous way to proceed was to use some form of computer aided software engineering (CASE) tool to capture the existing software requirements. This exercise could also be used to ensure that all the software requirements were captured in such a way that the subsequent design and implementation of the Ada solution mirrors the existing software. This will thus provide reliable comparisons between the two systems. There are many methodologies

available that come under the umbrella of requirements capture or analysis and design methods, such as CORE, MASCOT, SSADM, OOD, Jackson etc, but we decided to use Yourdon[1]. This choice was based mainly on two factors. Firstly our knowledge and experience with Yourdon over several years and secondly the availability of in-house tool support for this methodology. We had available in-house the CASE tool Teamwork[2] which implements a Yourdon based system analysis methodology and also has support for structured design.

4.1.1 System Analysis

The first task in the requirements capture was the analysis of the system. This is achieved in the Yourdon methodology by creating the top level context diagram. This defines the inputs and outputs of a system and thus places bounds on the extent of the system. This top level diagram was defined by searching through the software requirements for inputs and outputs. This task was aided by the fact that the hardware devices of the EEC were fixed and thus could be tied to individual software input and output functions. To simplify the diagram the various inputs and outputs were then collected together into logically functional blocks. e.g. collection of all engine input data, speeds, temperatures, pressures etc. into one functional block. The aim was to generate functional blocks that could be identified with individual system components. e.g. engine data, cockpit signals, airframe signals, fuel valve etc..

The resulting context diagram is shown in figure 4.1.

4.1.2 System breakdown

The next phase of the requirements capture process was to breakdown the context diagram, through several steps, into smaller, and logically independent, functional tasks. The first stage of this process was straightforward. As the EEC performs two different functions, dry engine control and reheat control, the first level of partition was to split the context diagram along this functional boundary. Then as the dry engine control consists of a dual lane system the next level of partition was to split the dry engine control function into the functions of the two lanes, termed lane A and lane B. As the lanes are functionally identical we then proceeded by continuing the partition for just one of the lanes.

The breakdown is performed under the Yourdon methodology by splitting an overall function into smaller and smaller component parts or processes as they are termed. This hierarchy consists of a set of what are termed dataflow diagrams. Each dataflow diagram consists of a set of process "bubbles", a set of input and output flows and a set of inter-process flows. Each process "bubble" can be split into component processes thus forming a new dataflow diagram. At each stage of the decomposition the data flowing into and out of a process must be maintained. To perform this task data composites are used to group together dataflow signals. A dataflow composite is simply a collection of dataflow items, which may be either elemental dataflows or other dataflow composites, that are grouped together under a single name. Thus at one level of the decomposition a process "bubble" as it is termed will have several dataflow items flowing into and out of it. When this process "bubble" is broken down into several component process "bubbles" the composite dataflows can also be split and each element associated with it's component process. In this way diagrams at the top of the hierarchy are not complicated by a mass of dataflow but by simple composite dataflows. Figure 4.2 shows the breakdown of the tasks for lane A.

For HOLD this process of gradually splitting the overall task into smaller and smaller items was terminated when further partitioning yielded no benefits. The decision as to where to stop the process was to a large extent arbitrary. The main goal was to reach a point that did not over or under partition a function. If the level is taken too low then individual functions could be fragmented and not easily assimilated. If the level is too high then functions will be too large and complex.

4.1.3 Process Definition

When the partition had been completed the end processes had to be defined. This was achieved using the process specification (P-Spec) feature of Teamwork. This allows a process to be described in terms of text and diagrams. The inputs and outputs are defined to this P-Spec automatically from the dataflow diagram. Figure 4.3 shows a low level dataflow diagram for a part of the control function of the EEC and figure 4.4 shows a typical P-Spec that we shall come across again later.

4.2 Structural Design

The task of structural design can be thought of as one of organisation. The objective is to take a set of specific requirements and organise them into coherent groups that will fit into the chosen hardware environment. If this is performed adequately then the task of the software design for the individual elements will, in concept, become trivial.

It is thus at this stage that the real world environment has to be considered. In essence the Yourdon analysis of the software requirements has no knowledge of the real time aspects of the engine

[1]Yourdon is a registered trademark of Yourdon Inc
[2]Teamwork is a registered trademark of CADRE Technologies Inc

control functions, or of any physical limitations such as size of memory available.

So the structural design was tackled with two different approaches. A bottom up approach to create the structure charts for the individual processes, and a top down approach for the executive structure controlling the order and sequence of execution of the processes.

4.2.1 Process Structure Charts

The functionality of individual processes within the requirements analysis was transferred to a structure chart format to represent the software design of each component. An example of a structure chart is shown in figure 4.5. These structure charts show the design tree of the software and how it is arranged into various blocks consisting of functional modules and data only modules. Each functional module is defined by a module specification (M-Spec).

The starting point for these M-Specs and data only modules is the P-Spec from the requirements analysis. The way in which a particular P-Spec is implemented will depend upon the language to be used in the implementation, as the M-Spec will have to directly reflect the requirements for the code. The data only modules may be defined directly from the input/output list of the P-Spec. Figure 4.6 shows the structure chart implementation of the P-Spec shown in figure 4.4 and figure 4.7 shows the M-Spec associated with this structure chart.

4.2.2 Executive Structure Charts

The executive structure charts fall into two types. Firstly there are the simple ones that reflect the dataflow diagram breakdown. Figure 4.8 shows the structure chart for the dataflow diagram shown in figure 4.3. This structure chart defines the calling sequence of the various modules which is not always obvious, or defined, in the dataflow diagram representation. This is an important feature which cannot be overlooked even though on the surface it seems to be a trivial task.

The second kind of structure chart is the overall executive which is typical of an engine control requirement. This is where real world detail has to be added in the form of power-up/initialisation requirements and iteration rates for the various processes. In the ideal world we would be able to specify the processing power requirements to run all the software functions together at the highest required rate but in the real world the processing power is often the limiting factor. This means that we have to divide the software functionality into elements which can be executed at different iteration rates so that only the most important functions are executed at the highest rate.

Figure 4.9 shows how this has been achieved for HOLD. This structure follows the original software structure which is only to be expected as the hardware is fixed. The main split is between power-up/initialise/base level functions which are one off or non time dependent tasks, and functions which are iterated at a fixed rate. There are two rates used, a fast level generated from a sample rate clock interrupt and a slower level generated at a multiple of the fast level.

It is this implementation of the software requirements into real iteration levels that causes problems with the structure charts. The example quoted in figure 4.4 is typical of the problem. The major portion of this function is executed at the slower rate but certain elements of it have to be iterated at the fast rate. This means that in designing the structure charts there is not a one to one relationship between a P-Spec and a structure chart.

4.3 Formal Methods Integration

In considering the application of Formal Methods to the HOLD programme, we decided that the best level at which to introduce such techniques would be at the P-Spec level.

We already had experience of the use of Formal Methods, in terms of the use of static analysis applied to small sections of assembler code. The next logical step was to move this process up a level to a small section of high level language code, and as such the P-Spec seemed the most appropriate.

There are many different functions within an engine control system and we decided to choose a representative sample of these functions for investigation. Four P-Specs were chosen. Two of these were engine control functions, one of which fell into the classical control area and involved lead-lag compensation, gain, lowest and highest wins elements etc., and the other fell into the logic area and involved sequencing functions. The third P-Spec was chosen from the area of signal validation involving range, rate of change and cross checks on a signal. The fourth function was selected from the area of the control associated with fault logging and diagnostics.

Our approach to the Formal Methods representation of these functions was firstly to select the language in which to represent them. There are several Formal Methods available and in choosing one we set out several criterion on which we based our selection. One factor in our choice was that the method should be widely accepted and supported. We wanted a method that was in popular use by the rest of the industry and one that was likely to stay in use for the foreseeable future. The method must also be supported in terms of training and course availability and

ideally would also have tool support available for automation of the Formal Method specifications and proof checking. For these reasons we chose the Vienna Development Method (VDM) which also had the added benefit of being the easiest to integrate with our existing systems.

The next step was to look at the implementation of the specifications from a general point of view. This activity led to the formulation of some general definitions which would be useful in the specification of discrete systems. These general definitions are based on the main feature of a discrete system, that is the periodic sampling of inputs and updating of outputs.

Such discrete systems operate cyclically, usually in responce to a sample rate clock interrupt, and prossess state variables which determine the operation of for instance timers, fault integrators and integrators. Formal definition of the outputs required where state variables are involved is best specified in terms of the history of the inputs, allowing a direct specification of the requirement to be made without stating the precise implementation ie what state variables are to be involved.

The history of an input is described via a map of cycle number to the signal's value. Note that not all cycle numbers need be represented in the domain of the map since an input may be read, for instance, on every second cycle.

Cycle Number = N;

Activation Cycle Set = Cycle Number-set

Booleans = Cycle Number $\xrightarrow{m}$ B;

e.g.

$hist_1$ = {1→false,2→false,3→true,...}
$hist_2$ = {3→false,5→true,7→false,...}

$domhist_1$ = {1,2,3,...}
$domhist_2$ = {3,5,7,...}

$hist_1(2)$ = false
$hist_2(5)$ = true
$hist_2(6)$ is undefined
$hist_2(6)$ = true i.e. the value of $hist_2(5)$

The application of this methodology may be better seen when applied to an example. Take for instance part of the P-Spec referenced earlier in figure 4.4 for the control of the IP blow of valve which states :-

> "The conditions for opening the IP Blow Off Valve are as follows:
> :
> :
> (1.0) Immediately on receipt of a Pilot_BOV_Signal, and held for a period of "N" seconds after the Pilot_BOV_Signal is removed."

Let BOV_Cycle be the set of cycle numbers at which the required state of the Blow Off Valve is determined. A function is needed which operates on the history of the Pilot_BOV_signal to produce a map from cycle numbers in BOV_Cycle to a value of true or false: true if an output value of true can be found in pilot_BOV_signal_HISTORY within the last "N" seconds, false otherwise.

```
BOVCheck:Booleans→Booleans

BOVCheck(PILOT_BOV_signal_HISTORY) Δ

  {c → ∃d ∈ dom PILOT_BOV_signal_HISTORY

        0≤ c-d ≤ ("N" * CycleFrequency)∧

         Pilot_BOV_signal_HISTORY(d)|

    c ∈ BOV_Cycle}
```

Cycle Frequency defines how many times per second a control cycle occurs. BOV_Cycle defines on which cycles the test is carried out. Thus with :-

```
CycleFrequency = 100

BOV_Cycle = {1,2,3,...}

Pilot_BOV_signal = {1→false,...,
                    50→false,...,
                    30000→true,
                    30001→false,...}
```

the example would yield a result of :-

```
{1→false,...,
50→false,...,
30000→true,
30001→true,...,
30000+"N"→true,
30000+"N"+1→false,...}
```

The VDM specification follows directly as:

```
ext   rd current_cycle       : Cycle Number
      rd Pilot_BOV_Signal_HISTORY: Booleans
      wr Lane_A_BOV_Control_Out  : B

post lane_A_BOV_Control_Out=BOVcheck
(Pilot_BOV_Signal_HISTORY)(current_cycle)
```

Since out implementation is based on a simple cyclic, scheduler what we require is an implementation that maintains a suitable loop invariant. However, we do not want to implement an operation which needs the complete history of the Pilot_BOV_Signal. We want an implementation that on each cycle calculates the result of the check for that cycle based only on the value of Pilot_BOV_Signal for that cycle and a small amount of additional state.

The obvious representation to use for the additional state is a counter whose value will be how many cycles ago the Pilot_BOV_Signal was last true. This counter will be called BOV_timer_count. In order to place a bound on the value of this counting when we reach the smallest integer which is larger than N*CycleFrequency. We will call this value BOV_End_Count.

```
BOV_End_Count:N=round(N * CycleFrequency+
0.5)
```

After identifying the loop invariant we can arrive at the specification of an operation which calculates the result of the blow off valve check. The role of this operation is to re-establish the loop invariant for each cycle.

```
ext rd current_cycle : CycleNumber
    rd Pilot_BOV_Signal_HISTORY:Booleans
    wr lane_A_BOV_Control_Out  : B
    wr BOV_timer_count     : N

pre 0 ≤ BOV_timer_count ≤ BOV_End_Count∧
    ~(∃ d ∈ dom Pilot_BOV_Signal_HISTORY•
    current_cycle-1-BOV_timer_count< d ≤
    current_cycle -1
    ∧ Pilot_BOV_Signal_HISTORY(d)) ∧
    BOVCheck(Pilot_BOV_
    Signal_HISTORY)(current_cycle-1)
    • (BOV_timer_count<BOV_End_Count ∧
    Pilot_BOV_Signal_HISTORY(current_cycle
    -1-BOV_timer-count))

post lane_A_BOV_Control_Out •
    BOVCheck(Pilot_BOV_Signal_HISTORY)
    (current_cycle)∧
    0 ≤ BOV_timer_count≤BOV_End_Count ∧
    ~(∃ d ∈ dom Pilot_BOV_Signal_HISTORY•
    current_cycle-BOV_timer_count
    <dscurrent_cycle
    ∧ pilot_BOV-Signal_HISTORY(d)) ∧
    BOVCheck(Pilot_BOV_Signal_HISTORY)
    (current_cycle)
     • (BOV_timer_count<BOV_End_Count∧
    Pilot_BOV_Signal_HISTORY(current_cycle
    -BOV_timer_count))
```

If we assume that there is a some mechanism to maintain the relationship:

```
   Pilot_BOV_Signal =
   Pilot_BOV_Signal_HISTORY(current_cycle)
```

then the implementation of the operations will only need to refer to Pilot_BOV-signal, and in fact the variables Pilot_BOV-Signal_HISTORY and current_cycle will not appear anywhere in our executable code (such variables are referred to as proof variables). The mechanism that maintains the relationship is, of course, the input routines.

From this VDM description the code may be produced, but before this step can be explained we have to look at the Ada programming environment.

4.4 Programming Environment

In considering the application of Ada to a real time engine control application there were two main areas that we had to investigate. Firstly there were the safety aspects of the language and secondly there were the timing implications.

4.4.1 Safety Critical Language Features

In the use of any language there are generally features of that language that are not desirable in a safety critical engine control software systems. As stated earlier the failure of the software to complete a function is a serious error not merely an inconvenience. It is of critical importance, therefore, that any function in the software has an explicit entry and exit condition. For this reason loop constructs of the type DO-UNTIL, DO-FOR and DO-WHILE are avoided especially where the loop parameter is determined at the run time of the system. For a similar reason the use of GO-TO type constructs should be avoided as they permit ad-hoc entry to and exit from functions.

Another major feature of safety critical engine control systems is that in general the hardware environment is composed of a custom made unit. This means that any software language used to program these units must have the ability to interface with the unique hardware of the unit. This is available with Ada, and with other languages, by means of an interface with assembly language components. The hardware constraints also limit in a system the amount of memory available and for this reason it is preferable to know in advance how much storage is required. Thus features which dynamically allocate memory at run time must be avoided. Ideally all the memory should be statically defined at compile or link time, or in the case of a stack for example the bounds of the memory requirements should be calculable.

4.4.2 Time Critical Language Features

A typical engine control system runs at an iteration rate in the region of 20 milliseconds, the time being chosen to achieve satisfactory engine control response. Thus time is critical in an engine control environment as there is no option available to increase the run time of the software. For this reason the efficiency of the language is of prime importance. From experience we knew that the run time system was going to be an important area to investigate, not only in respect of Ada itself but also in respect of the particular compiler selected.

We required a run time system that could provide a simple and quick interrupt transfer mechanism without the use of tasking because of the time response associated with this feature of Ada.

Another feature of Ada that we wished to avoid was the use of exception handlers. The main source of exceptions in an engine control system is due to integer overflow. As integer operations are widely used this

would need in theory an exception handler for each operation as the result required would be different in each case. This approach would place a unacceptable overhead on the execution time of the code. The preferred method is to design out or protect against overflow conditions, so that exception handlers become redundant.

4.4.3 Compiler Selection and Restrictions

As a result of the requirements set out in the previous two sections we selected the Ada compiler from SD-Scicon since it has a minimal run time system, which could also be tailored to our own individual requirements. The selection was also influenced by the needs to operate on our own computer system in terms of target/host configuration.

In considering the possible need to apply restrictions to the Ada language we looked for a way of providing a safe sub-set of the language. Ideally we required some way to automatically test code for illegal construct usage. There is a very limited set of products in this field but one which fitted not only our requirements for a safe sub-set, but also our needs in the integration of Formal Methods, was SPARK (SPADE[1] Ada Run-time Kernal).

SPARK is a tool which checks Ada source code for a variety of restricted features and issues warnings if any code violates these conditions. The tool also performs tasks associated with the static analysis of the code. This provides flow checking of the code as well as verification condition generation and proof checking. These last two elements fitted well with the integration of Formal Methods in the generation of the code. We could thus use the VDM specifications to generate pre and post conditions in the Ada code which we could then prove using SPARK.

5. INITIAL CONCLUSIONS

At present the HOLD programme is approximately 75% complete. The analysis and capture of the software requirements and the software design phases are essentially complete. The main activities at the moment are the coding and the generation of the VDM specifications.

5.1 Software Requirements Document

The analysis of the software requirements document using the Yourdon methodology has resulted in a very easy to read document. The way that the requirements are split down into finer and finer detail means that at each level of the hierarchy of dataflow diagrams a comprehensible amount of information can be given. It must be remembered, however, that the methodology only serves as a tool to represent the software requirements, it only helps to specify the requirements in so far as laying them out in a clear manner so that ideas can be communicated easily.

5.2 Design Description Document

As is the case with the software requirements, the use of structure charts generated from the dataflow diagram breakdown has provided a clear description of the software design. The main problem, as with all systems, is that of the transition from what is required to how it is to be implemented. The addition of implementation dependent features at the software design stage can obscure the flow of information from the analysis to the design phase. This problem can be overcome, however, if the analysis is performed with the implementation in mind. In the life cycle of any project there is rarely just one iteration around the loop from requirements to design to code. In practice this loop is iterated around many times. The software requirements at the beginning of a project are seldom complete and develop over the life cycle of the project. In this way the Yourdon breakdown may serve initially as a definition of the requirements but as time progresses this definition can be developed so that the requirements are broken down in a manner which suits the chosen implementation. This will lead, eventually, to a closer one to one relationship with the design phase and thus simplify the software development process.

5.3 Ada Run Time System

Our work done on the assessment of run time systems suitable for this application, as part of the selection of an Ada compiler has shown that the area of "bare micro" targets is being addressed by compiler vendors. As recently as five years ago it would have been impossible to purchase anything other than a large run time system aimed at a large computer system target. Now more attention is being focused on almost "bare micro" target systems. The fact that we have been able to take an off the shelf system, albeit with some tailoring, is testimony to this development. Using this supplied system we have so far developed a bare run time system (interrupt servicing and test port communication) which is working in the target unit.

The timing and memory utilisation of this system has been shown to be acceptable for engine control applications.

5.4 Comparisons with Traditional Methodology

The traditional development of software has relied on English language descriptions for the software requirements and design. Whilst this is still true in part for HOLD, as most of the P-Specs are still in English language, the application of the new methodology has served to present the requirements and design in a clearer form. The work in hand at present

[1]SPADE is a registered trademark of Program Validation Ltd

on the application of Formal Methods to the P-Specs will show whether we can replace the English language descriptions with a rigorous mathematical description.

6. FUTURE WORK ON HOLD

Once the work of completing the coding and Formal Methods implementation of the selected P-Specs is complete the main task of qualitative and quantitive assessment can begin.

We have already been able to assess the implication of using Yourdon and the effects of the Ada run time system and both have shown positive results for the future of engine control software development. The areas that have yet to be assessed in the future are threefold. Firstly there is the assessment of the Ada code itself. Comparisons will be made between the Ada and original LUCOL Software code. These comparisons will address not only the efficiency of the two languages, run time, memory utilisation etc. but also the speed and ease of code development. Secondly there is the analysis of the effects that the use of Formal Methods will have on software development. Our work to date has shown that application of Formal Methods should lead to unambiguous specifications. However, the practical application of such methodologies is not easily attainable by engineers, because of the highly mathematical nature of the system plus the fact that experience of their use in real word situations of this type is extremely limited. Thirdly there is the aspect of validation of the system to be considered. In the past critical engine control software has been verified down at the level of the target code resident in the target environment. Using Formal Methods, for example, will allow the mathematical proof that a piece of Ada code meets its formal specification. This proof relies on the Ada compiler producing correct code. Whether this is an acceptable system for validating software has yet to be assessed.

6.1 Conclusions of Project

The work completed so far on the HOLD Project has emphasised that the application of Ada, formal notations and CASE tools to the flight safety critical military gas turbine FADEC brings particular problems.

However, we are also beginning to see the potential benefits within the total process that the application of a structured, system level approach can bring. It appears that if such an approach is applied throughout the process, from requirements capture through to implementation, that:-

i) the likelihood of specification errors will be reduced,

ii) the likelihood of misinterpretation of functional requirements will be reduced,

iii) the interface responsibilities between the various partners should be more easily identified,

iv) The functional interface between the elements of the system should be more easily identified,

v) programs monitoring and the process to clear the system for flight should be less difficult to manage,

vi) the whole process will be more effective in the use of all resources deployed.

HOLD will enable the military aero-gas turbine community to make a positive contribution to the general understanding of the topics that are being studied. We will more clearly understand where it is appropriate to use generalised techniques and where we must be concerned because of the application specific constraints. Where we do identify clearly that a FADEC does require particular considerations then we will be able to present cogent reasons for those considerations and influence future standards and working practices.

HOLD will improve our ability to identify programme technical and financial risk and hence improve UK MOD(PE)'s ability to procure functionally capable systems on-time and on-cost.

7. ACKNOWLEDGEMENTS

The authors would like to thank MOD(PE) and Lucas Aerospace Limited for their permission to present and publish this paper.

The authors would also like to place on record their appreciation for the support of their colleagues in preparing this paper and for their contribution to the HOLD project. This also applies to other companies involved in the HOLD project, in particular Program Validation Limited.

The opinions expressed in this paper are entirely those of the authors and do not necessarily represent those of their respective organisations.

This work has been carried out with the support of Procurement Executive Ministry of Defence.

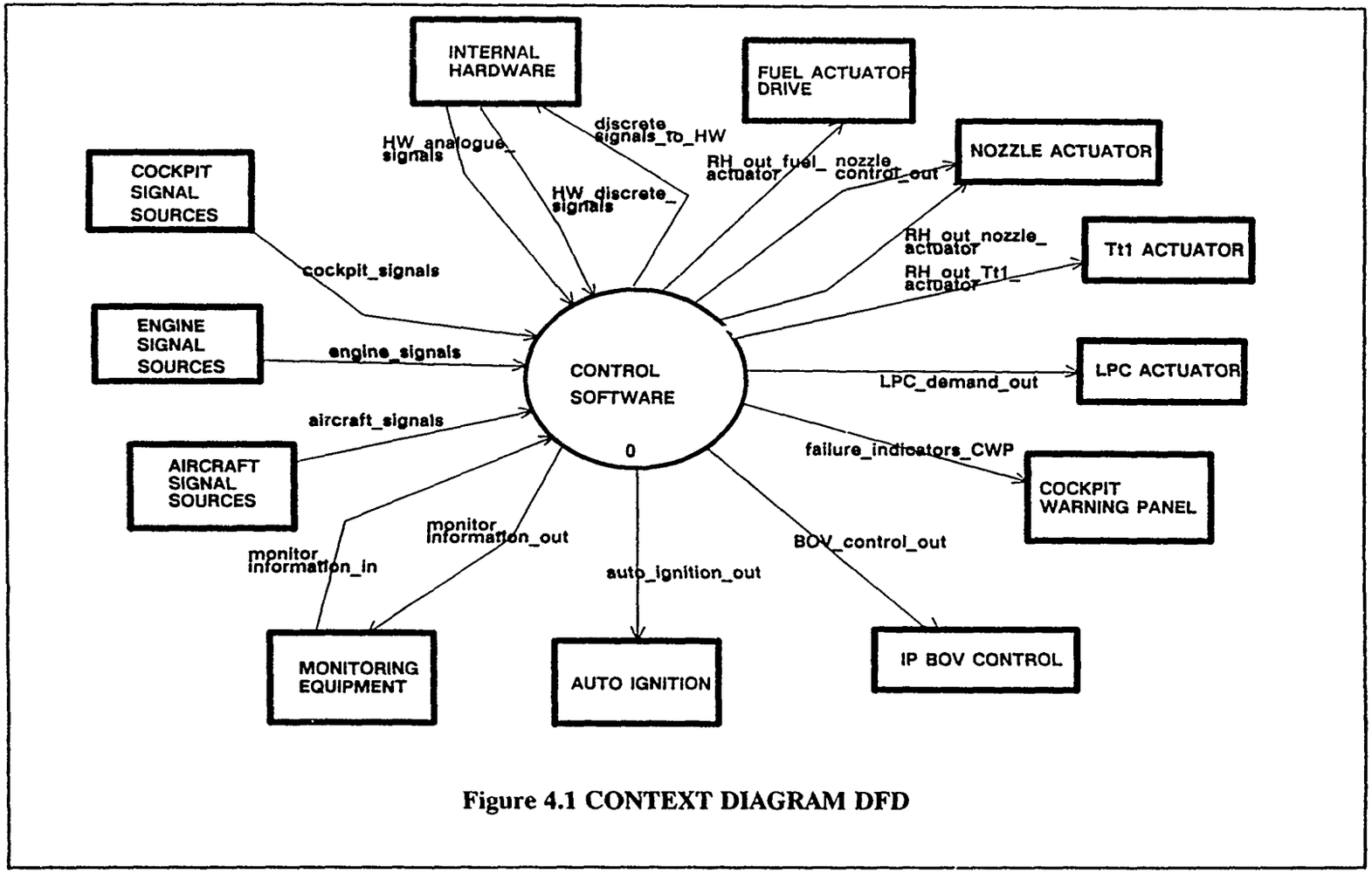


Figure 4.1 CONTEXT DIAGRAM DFD

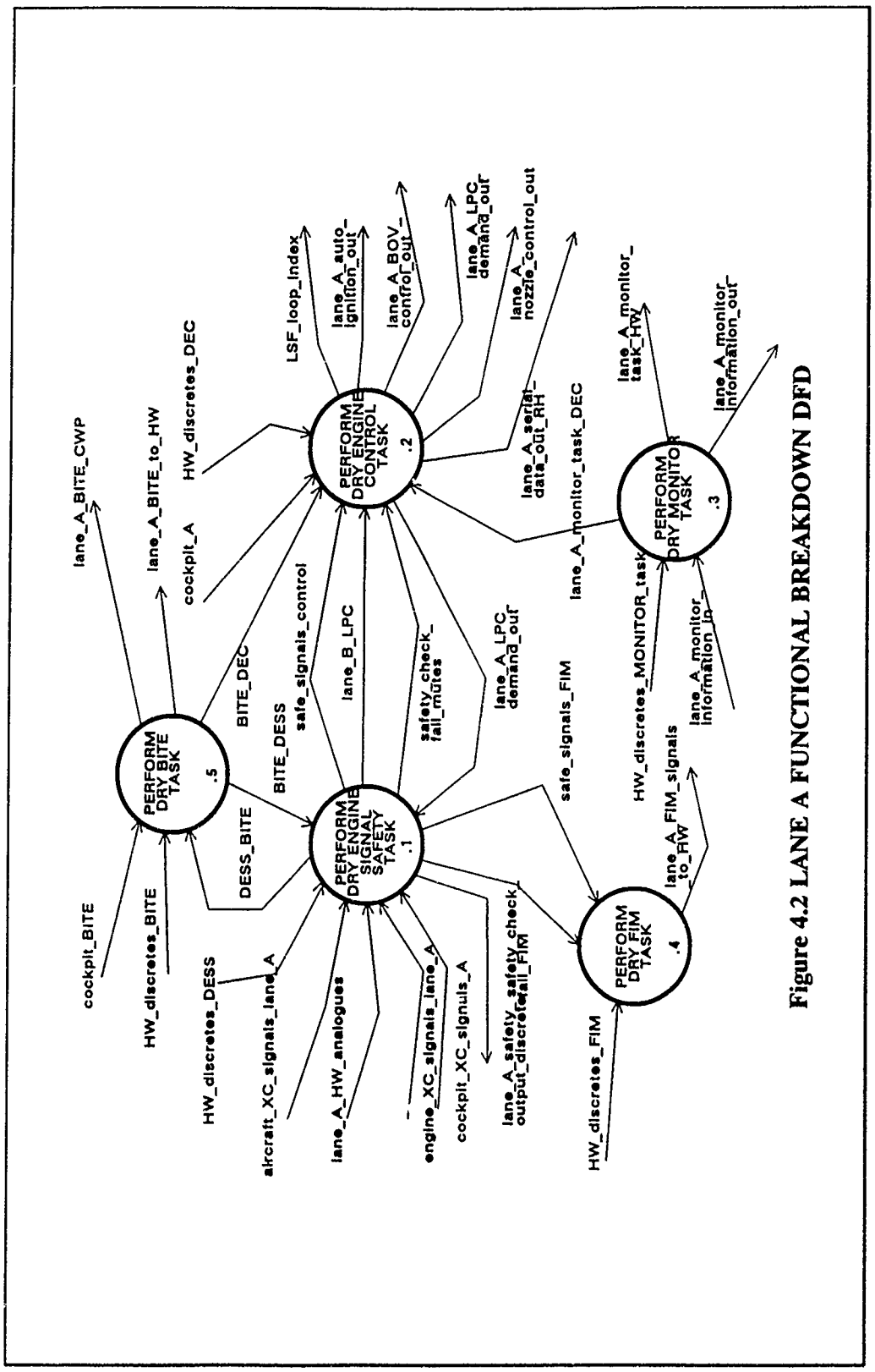


Figure 4.2 LANE A FUNCTIONAL BREAKDOWN DFD

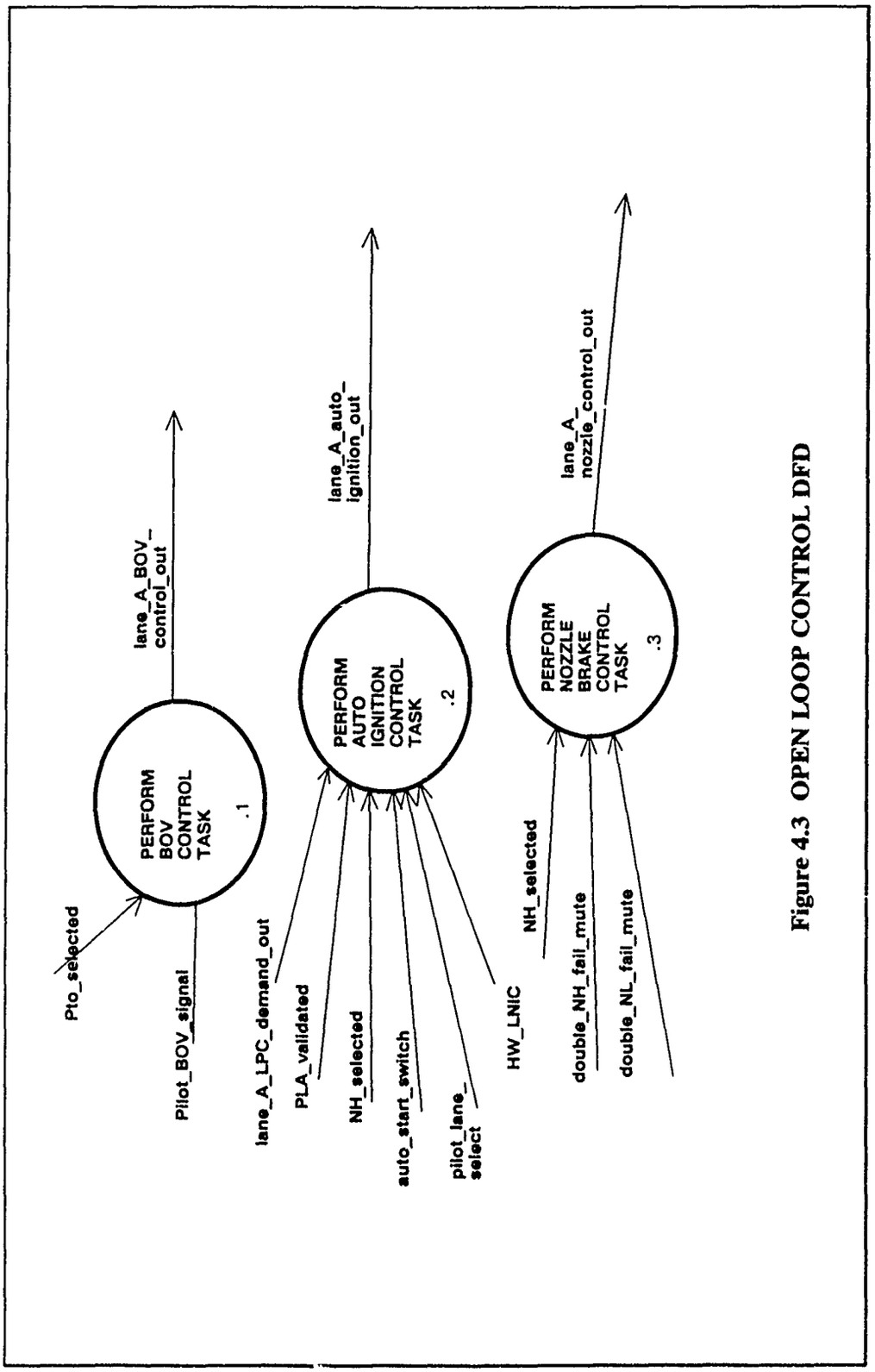


Figure 4.3 OPEN LOOP CONTROL DFD

NAME:
1.1.2.1 1;1

TITLE:
PERFORM IP BOV CONTROL TASK

INPUT/OUTPUT:
Pto_selected : data_in
Pilot_BOV_signal : data_in
lane_A_BOV_control_out : data_out

BODY:
Purpose.
/1
To control the operation of the IP Blow Off Valve.

Function.

The conditions for opening the IP Blow Off Valve are as follows:
1.0 Immediately on the receipt of a Pilot_BOV_signal, and held for a period of "N" seconds after the signal is removed.
OR
2.0 The following Pressure conditions are achieved
Pto < x Kpa Pto decreasing
Pto < y Kpa Pto increasing

Figure 4.4

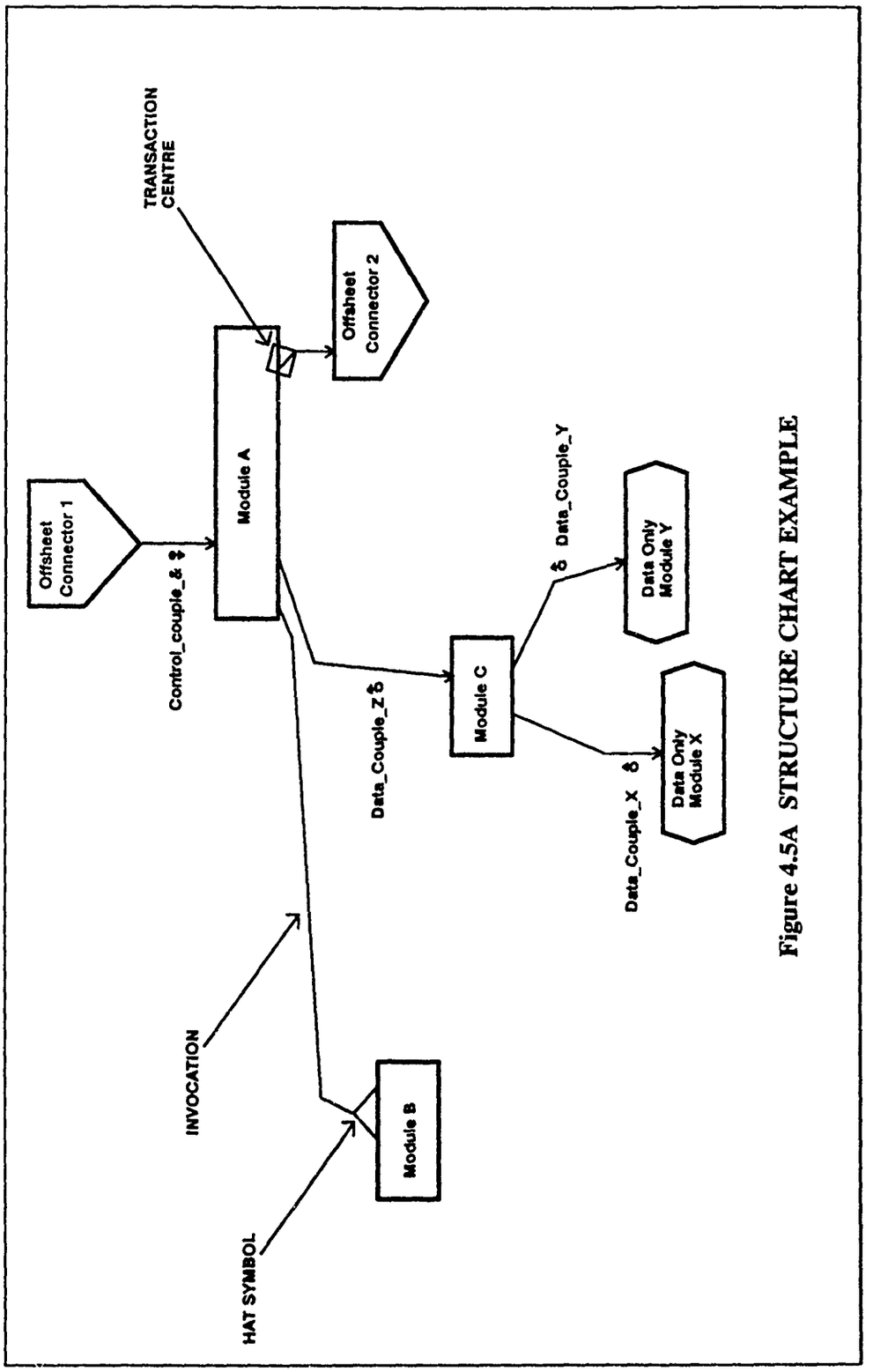


Figure 4.5A STRUCTURE CHART EXAMPLE

1 STRUCTURE CHART EXPLANATION

Introduction

Each structure chart is composed of an arrangement of various graphic symbols these symbols are described below.

1.1 Module MSPEC

Modules A B and C are Plain modules,they represent some detailed processing activity the name of the module relates to the nature of the activity occurring within the module.
Associated with each module is an MSPEC which details the procedural aspects and actions performed by the module.

1.1.1 Data Only Module

The Data Only Module represents Global data, that is data used Globally throughout the software, and each data only module can represent a single instance of an item of data, or an aggregate of such items stored in a particular location.

1.1.1.1 Offsheet Connector

The Offsheet Connector is used to represent the existence of a further structure chart, and allows decomposition of more complex charts into simpler subordinate structure charts.

1.1.1.2 Data Couple

The data couple represents data flow within the structure chart,they must be attached to an invocation. The couple can represent Global or Local data, the direction of the arrow represents the direction of the flow.

1.1.1.3 Control Couple

The control couple represents control flow within the structure chart,they must be attached to an invocation. The couple can represent Global or Local control, the direction of the arrow represents the direction of the flow.

1.1.1.4 Transaction Centre

Represents some decision making process, such as conditional invocation of a subordinate module.

1.1.1.5 Hat

Represents Textual inclusion, that is the body of the MSPEC and the action performed by it is meant to be included within the calling module.

Textual inclusion only occurs between plain modules, and does not occur between offsheet connectors and or data only modules.

1.1.1.6 Invocation

The invocation line is symbolic of a call from a module to other symbolic items.
It should be noted that the philosophy adopted throughout this document is that only modules can invoke other symbolic items.

The offsheet connector is used to connect invocations from a module to another structure chart, in this case the interface between the corresponding connectors on each sheet must match exactly.

Figure 4.B

The **precedence** of invocation is **Left to Right and Down,** with reference to Fig 1.

Offsheet connector 1 with a control parameter invokes

Module A
The module performs its required operation

Module A
Invokes
Module B
The invocation is terminated with a hat,
The module performs its required operation
Module A
Invokes
Module C which invokes (reads or writes) from the data only modules X and Y, performs some action and then returns the data couple Z

Module A conditionally invokes
Offsheet connector D which performs some action .

Figure 4.5C

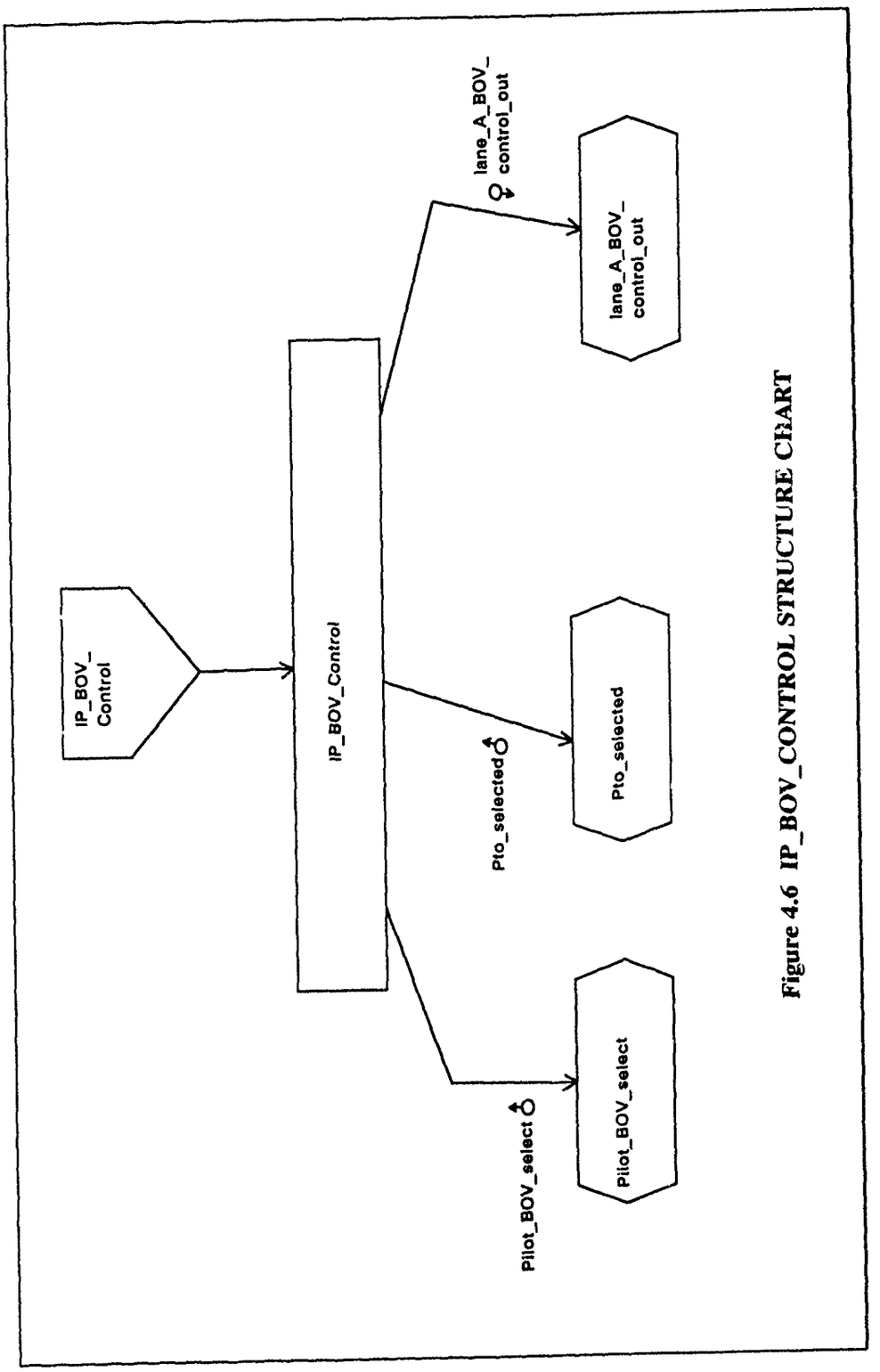


Figure 4.6 IP_BOV_CONTROL STRUCTURE CHART

NAME:
IP_BOV_Controi;12

TITLE:
IP Blow off valve control procedures

PARAMETERS:

LOCALS:
Pre_condition_X
Pre_condition_Y
Pto_last
Pto_limit_1
Pto_limit_2

GLOBALS:
Pto_selected : data_in
Pilot_BOV_signal : data_in
lane_A_BOV_control_out : data_out

BODY:
Purpose/Description

IP_BOV_Control is a subordinate MSPEC on the Structure Chart of the same name.
It is invoked via the offsheet connector of the same on the Structure Chart
Open_Loop_Control.

1)Read in the Globals listed above in the GLOBALS list as data_in.
2)Evaluate the conditions for opening the IP BOV.
2.1 Pressure Limits.
  Pto_limit_1 = x Kpa
  Pto_limit_2 = y Kpa
2.2 Evaluate Pre_condition_X
    If ((Pto_selected < Pto_limit_1) and (Pto_selected < Pto_last) then
      set Pre_condition_X = TRUE
    else
      set Pre_condition_X = FALSE
2.3 Evaluate Pre_condition_Y
    If ((Pto_selected < Pto_limit_2) and (Pto_selected < Pto_last)then
      set Pre_condition_Y = TRUE
    else
      set Pre_condition_Y = FALSE
2.4 Determine new condition for lane_A_BOV_control_out
    If (Pre_condition_X or Pre_condition_Y or Pilot_BOV_signal) = TRUE then
      set lane_A_BOV_control_out = TRUE
    else
      set lane_A_BOV_control_out = FALSE
3) Pto_last = Pto_selected
4) Write out the Global lane_A_BOV_control_out

Figure 4.7

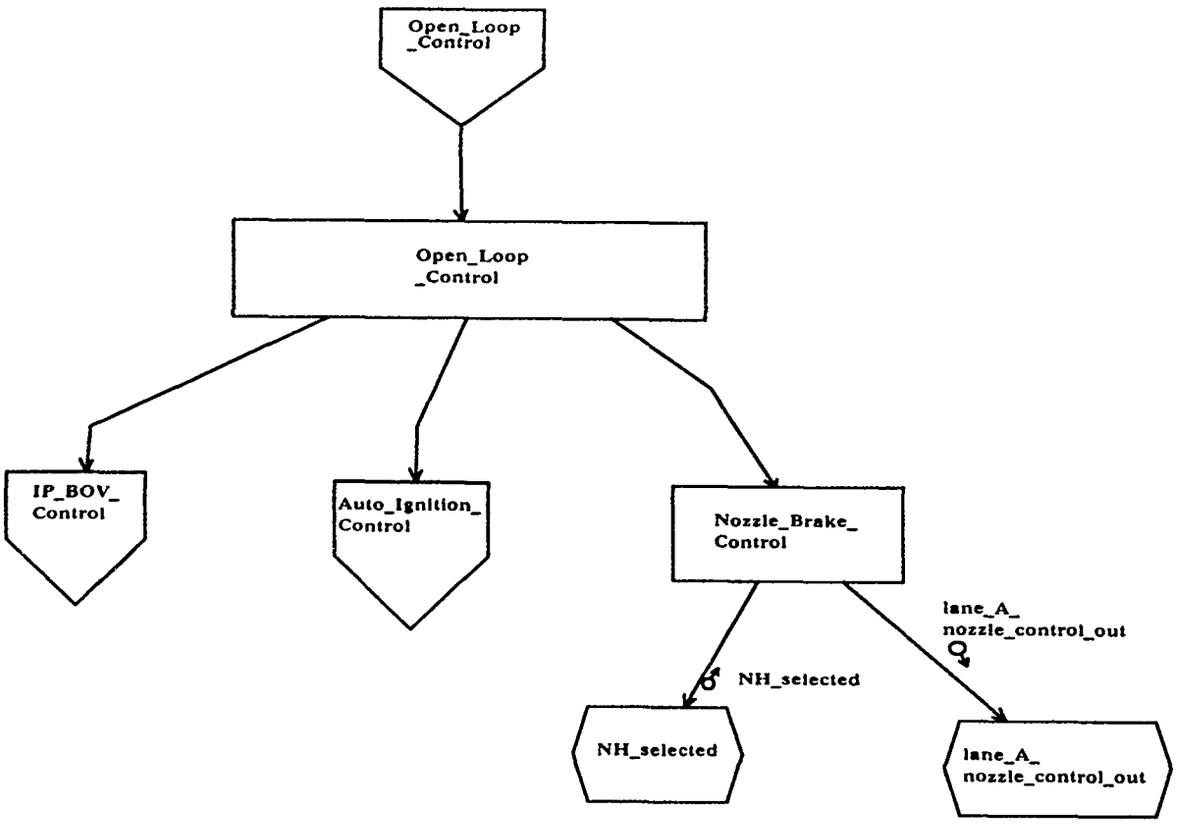


Figure 4.8 OPEN_LOOP_CONTROL STRUCTURE CHART

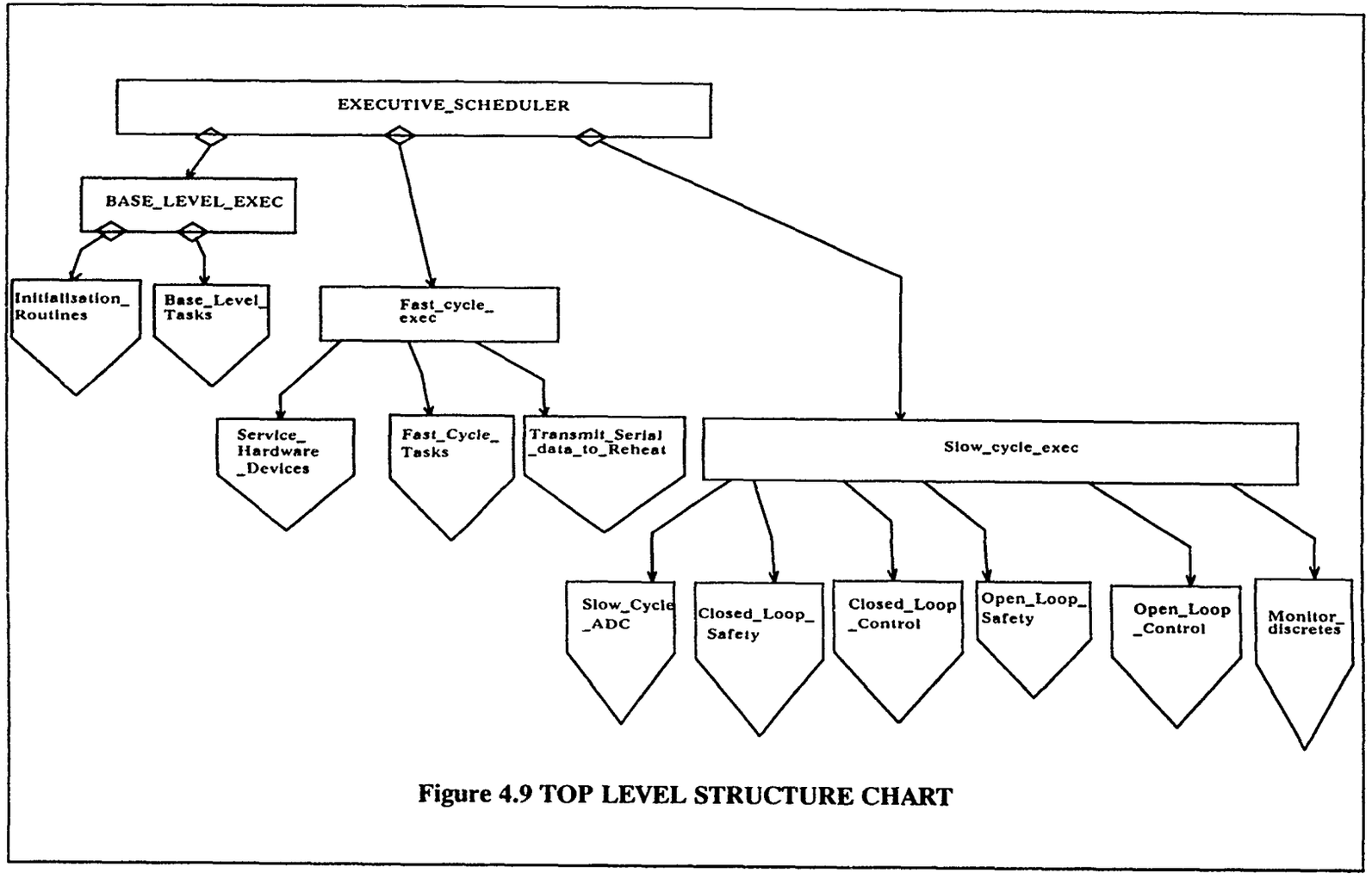


Figure 4.9 TOP LEVEL STRUCTURE CHART

COMMON ADA MISSILE PACKAGES (CAMP)

Barry E. Mullins
Armament Directorate
Wright Laboratory
WL/MNAG
Eglin AFB, Florida 32542-5434

Abstract

The words "software crisis" should not be new to anyone managing a program that involves software. A shortage of skilled, software personnel is adversely affecting the development and subsequent maintenance of today's and future weapon systems. The Department of Defense (DOD), as well as industry, acknowledge this crisis and are taking bold measures to alleviate it. The Common Ada Missile Packages (CAMP) program is one such measure the DOD has undertaken to ease the crisis via a high-payoff remedy -- reuse of real-time embedded (RTE) software. CAMP is a pathfinding effort designed to investigate the feasibility of RTE software reuse by actually developing reusable Ada parts, compiler benchmarks and a parts engineering system (PES). This paper describes the genesis of CAMP, structure of the CAMP program, evaluation results and CAMP products. McDonnell Douglas Missile Systems Company developed the CAMP products under the sponsorship of the Armament Directorate, Wright Laboratory at Eglin Air Force Base, Florida.

The Software Crisis

The amount of software required to operate weapon systems over the past 30 years has grown tremendously. Where earlier F-4 fighter jets had no software systems, today's B-1 bomber is saturated with well over 1 million lines of code. This is just one example of the insatiable demand for complex software systems which are typically the major cost driver of weapon systems. This demand has not been matched by the education and acquisition of skilled software developers and maintainers. This imbalance ultimately resulted in the use of out-dated software methods and tools being applied to highly complex, sophisticated applications thus leading to sometimes inferior, unreliable weapon systems being delivered late and almost always over budget.

In 1983, the Air Force Software Technology for Adaptable Reliable Systems (STARS) Task Force published a report on the software crisis and possible solutions. The report recommended software reuse to ease current software problems. Software reuse by itself is not the panacea to the software crisis but seems to offer tremendous returns. Potential benefits include increased software development productivity of more reliable software systems and more efficient use of software engineering expertise. Furthermore, in 1983, the DOD mandated the use of the Ada programming language (ANSI/MIL-STD-1815A) in all new embedded systems. Since the constructs of the Ada language are extremely amenable to reuse, the Air Force Armament Laboratory (now the Armament Directorate, Wright Laboratory) initiated the CAMP program to investigate software reuse for conventional missiles using Ada (what else?).

The CAMP Solution

Although software reuse has been practiced with varying levels of success prior to the STARS report and the Ada mandate, it was almost always ad hoc reuse and the application domains were typically not as constrained by size and speed as found in the conventional missile arena. It should also be noted that prior to Ada, programming languages were not equipped with the necessary facilities to directly support software reusability nor were they as highly standardized (i.e., supporting software transportability between platforms).

The CAMP program was designed to address these limitations. CAMP focused on three primary areas as they relate to operational missile flight software: (1) investigate the feasibility and applicability of software reuse; (2) design and develop reusable Ada parts, Ada compiler benchmarks and a supporting environment for the Ada parts; and (3) refine, productize and transition the technology. To satisfy these goals, CAMP was performed in three separate contracts all competed and won by McDonnell Douglas Missile Systems Company. The contracts satisfied a particular phase of the program and were therefore called phases.

CAMP-1: Feasibility Study

The first phase of the CAMP program, Phase 1 (CAMP-1), began in September 1984. CAMP-1 was a one year feasibility study designed to determine the scope of commonality among missile flight software. Assuming sufficient commonality existed, the top-level design for common parts would be developed. Also, the feasibility and value of automating the process of building these software systems using parts was to be investigated.

Before further discussion, it is important to fully understand what constitutes a CAMP part. The CAMP program used the following definition: A part is an Ada software package, subprogram or task that must be usable in a stand-alone fashion (i.e., does not depend on external code for proper execution). However, parts may "with" other parts. The goal of CAMP-1 was to develop elementary, flexible parts which provide a useful function to more than one application while maintaining run-time efficiency.

Domain Analysis

The feasibility study included a domain analysis that attempted to identify common operations, objects and structures within a bounded domain. Although a domain analysis is expensive and laborious, it is imperative to verify domain commonality exists before any attempt is made at designing reusable parts. To attempt parts development without a domain analysis would be a waste of effort since the resulting parts would not offer true commonality within the application area.

Ten existing missile systems were included in the domain analysi which included at least two missiles from the following classes: air-to-air, air-to-surface, surface-to-air and surface-to-surface. By studying documentation and source code for the missiles, sufficient commonality was verified to warrant the design of parts.

During the domain analysis it was discovered that missile domain parts can be properly separated into two types -- domain dependent and domain independent. Domain dependent parts are applicable only to the missile flight software domain. Domain independent parts can be used in other domains with few, if any, changes. An example of domain independent parts include the mathematical parts which are essential to missile software but also may be used in other areas.

CAMP-1 successfully demonstrated commonality existed within the missile domain. A total of 219 reusable parts were identified. The requirements and top-level design of each part were documented, and a software parts taxonomy was created to facilitate parts classification and organization. Part complexity ranged from simple mathematical functions to complex processes and structures.

Parts Engineering System and Cataloging Scheme

The development of efficient reusable parts is a major milestone in the fight against the software crisis. However, parts alone are not the answer; tools must be developed to organize, index, describe and reference the parts to fully exploit software reuse. Therefore, substantial effort was invested in the CAMP tools. In CAMP-1 a top-level design for a Parts Engineering System (PES) was developed.

The ultimate goal of the PES was to facilitate storage and retrieval of relevant software parts for use on other projects while increasing the productivity of the parts user. The development of the PES included the investigation of cataloging schemes and documentation requirements. The actual capabilities of the PES are discussed in the CAMP-2 section.

The candidate cataloging scheme for CAMP was studied in great detail. Realizing the role effective catalogs play in successful software reuse, the study included research into existing catalog schemes and philosophies. Without an adequate cataloging scheme, the identification of particular software parts becomes cumbersome at best and in some instances virtually impossible. A successful cataloging scheme must include sufficient information to determine applicability of parts to the user's domain/problem and efficiently retrieve parts without burdening the user with too much data. An overload of information can be counterproductive and ultimately lead to the failure of the system due to lack of use. Figure 1 displays the catalog attributes utilized in the CAMP PES catalog.

A usable catalog system must take yet another step to ensure success of the software reuse effort -- complete documentation of the software parts. Every part must be thoroughly documented with standard data to provide future users with necessary decision-making information. A standard form should be developed for the catalog entry effort to ensure all necessary information is supplied when a part is entered into the catalog. In the future, on-line data entry may eliminate the need for such a form.

ABSTRACT	ORIGINAL DATE OF CATALOG ENTRY
BODY FILES	PART NAME
CATALOG ENTRY REVISION DATE	PART NUMBER
COMPILATION INSTRUCTIONS	PERFORMANCE NOTES
CONSOLIDATED TEST CODE FILE	REQUIREMENTS DOCUMENTATION
DEPENDENCIES	RESTRICTIONS
DESIGN DOCUMENTATION	REVISION HISTORY
DESIGN ISSUES	REVISION NUMBER
DEVELOPER	SAMPLE USAGE
DEVELOPER COMMENTS	SPECIFICATION FILE NAME
DEVELOPMENT DATE	STATEMENT COUNT
DEVELOPED FOR	TAXONOMIC CATEGORY
GOVT SENSITIVITY OF ENTRY/PART	TYPE
KEYWORDS	USED BY
LINES OF CODE	USER COMMENTS
ORGANIZATIONAL SENSITIVITY OF ENTRY/PART	WITH'D BY
	WITHS

Figure 1 CAMP PES Catalog Attributes

Automated Software Generation Using Parts

To facilitate the development of missile software systems, a study into the feasibility and value of developing an automated means of generating software using existing parts was performed.

The concept of automatic software generation is not new. In fact, from the dawn of machine language, researchers were devising mechanisms t automate software generation by abstractio coding. At the time, they called this assembl language. As software technology advanced, so did researchers' expectations. They continued to expect automatic software generation capabilities -- hence the birth of higher order languages. Today, the quest is turning towards VHOL (Very High Order Languages). VHOL allows the user to enter specifications and requirements at a high level of abstraction.

The reward for automating software generation is low-cost, quality software via reduced development time and cost. This is attained by requiring less detailed design knowledge of software developers. Thus, a domain engineer would be able to directly develop a software system by inputting his domain requirements into the generation

system thereby bypassing the software engineer altogether resulting in overall cost savings. These savings are enhanced by the use of existing parts. In addition to the cost savings, the parts offer greater reliability since they have been previously tested.

After studying various automatic software generation systems available at the time, specification techniques, methods of operation, text generation and expert system assistance, the CAMP "constructors" were designed. Constructors are software templates that when combined with user input to customize the template results in the generation of complex software components. Constructors are supported by the PES expert system and a limited natural language interface.

CAMP-2: Development Effort

The beginning of CAMP's second phase (CAMP-2) coincided with the conclusion of CAMP-1 in September 1985 and finished 32 months later. The primary goal of CAMP-2 was to complete the development of the CAMP software and demonstrate CAMP technologies in a credible, demanding application. This included the development and testing of the reusable Ada parts, the parts engineering system and Ada compiler benchmarks and the use of this software to build an "11th missile" (so named because it was not in the original domain study from which the parts were generated).

Development and Testing of CAMP Parts

The requirements and designs developed during CAMP-1 formed the foundation of the actual part implementations during CAMP-2. All 219 parts identified during CAMP-1 were coded, tested, and documented during this phase. While developing these parts, an additional 235 parts were identified, designed, coded and tested during CAMP-2 boosting the total number of parts to 454.

One of the design goals for the parts was to keep them simple thus facilitating understandability and reuse. Part simplicity was enhanced by keeping their size small. The parts ranged from 10 to 100 Ada statements. Another way to enhance part simplicity was to ensure the granularity of the parts were at the lowest possible level. In other words, the missile tasks were broken down into the lowest possible functions while maintaining understandability of the part. Also, complex parts were developed using a combination of simple parts.

Another factor leading to the successful reuse of software is adequate documentation. The CAMP parts were documented extensively. Unfamiliarity of the parts is the primary purpose for this documentation. A new user of the CAMP parts will be unfamiliar with them and require tremendous information on their operation. Also, CAMP parts use Ada's "generic units" which may be foreign to most users. The documentation provides the necessary information, including samples, to properly instantiate and use the parts.

During parts development, effort data were carefully tracked to determine productivity. To ensure an accurate and fair comparison with other efforts, two metrics were used to calculate the size of the parts -- lines of code and Ada statements. A line of code was defined as any line in the source code file which contained all or part of an Ada statement. An Ada statement count is simply the number of Ada statements in a source file (i.e., the number of semicolons). Figure 2 illustrates the total size of the CAMP parts. The figure also reveals the enormous amount of documentation -- 9 comment lines per Ada statement.

	LINES OF ADA CODE	ADA STATEMENTS	LINES OF COMMENTS
PART CODE	18,091	10,203	91,553
TEST CODE	27,584	17,991	
TOTAL	43 675	28,194	

Figure 2. CAMP Parts Sizing Data

Figures 3 and 4 provide the development productivity and statistics respectively. As the figure illustrates, the productivity experienced during the CAMP parts development was 61% greater than the predicted value from the COCOMO model. Factors leading to this increase included the use of the Ada language, well-trained people, good tools and code reuse. Specifically, Ada's attributes (e.g., strong data typing) contribute to increased productivity by allowing early detection of errors. The CAMP team had some Ada experience prior to the program and received training in software engineering practices. Utilizing software engineering tools also increased productivity. Finally, productivity was increased by reusing CAMP parts during the development of other parts.

	PART CODE ONLY		PART & TEST CODE	
DESIGN-TESTING EFFORT	LOC/MM	383	LOC/MM	1039
	STMT/MM	243	STMT/MM	671
	MH/LOC	0 407	MH/LOC	0 150
	MH/STMT	0 643	MH/STMT	0.233
ALL EFFORT	LOC/MM	258	LOC/MM	750
	STMT/MM	184	STMT/MM	452
	MH/LOC	0.605	MH/LOC	0.223
	MH/STMT	0 954	MH/STMT	0.345

Figure 3. CAMP Parts Productivity Data

The Parts Engineering System (PES)

With feasibility established and the requirements and design completed during CAMP-1, a prototype PES was coded, tested and documented during CAMP-2. The PES consisted of three integrated subsystems designed to provide expert assistance to the user: a parts catalog, a parts exploration system and component constructors.

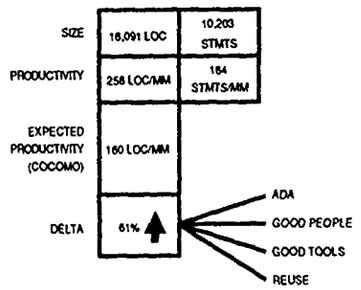


Figure 4. CAMP Parts Development Statistics

Catalog Subsystem

An extensive cataloging capability should be the backbone of any parts engineering system. The CAMP PES is no exception; the catalog subsystem is the foundation of the PES. The goal of the parts catalog is to help the PES user to clearly understand the Ada parts and facilitate their efficient retrieval and reuse. The catalog allows the user to add, modify or delete reusable software entries. It also provides the following functions: searching for catalog entries based on various attribute values, examining both catalog entries and Ada part source code, and generating printed versions of the catalog entries.

Exploration Subsystem

The exploration function provides the user, typically a missile system engineer or a missile software requirements engineer, with the ability to identify potentially applicable parts for a software system. The primary difference between the cataloging function and the exploration function is the latter deals with a higher level of abstraction. The exploration function allows the user to specify requirements while not concerning himself with part specifics. This function actually maps the missile system requirements to the parts. Therefore, it is designed for use early in the development cycle (i.e., requirements/design phase) to drive the design towards maximum software reuse. Used early in the development, the function assists software cost estimate, sizing and timing studies, and make-or-buy trade-off studies.

The PES uses two techniques for parts exploration. The first is the application approach and is designed to map high-level system requirements to existing parts. Through a series of questions, the application approach generates a list of potentially applicable parts. Figure 5 depicts the various types of information requested, as well as the selected parts.

The second exploration technique is the architectural approach which allows the user to walk through a hierarchical model of missile flight software. The models were developed using knowledge of the various missile systems and depict the subsystems, functions, and applicable CAMP parts. Based on user inputs, the appropriate model is selected for further user examination and subsequent part exploration. Figure 6 illustrates the architectural approach.

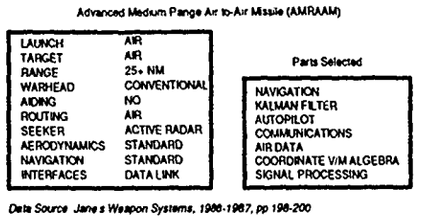


Figure 5. Application Exploration Example

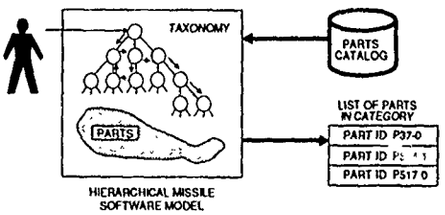


Figure 6. Missile Model Walkthrough (Architectural Approach)

Component Constructor Subsystem

The objective of the component constructor is to generate application-specific, tailored Ada code based on user requirements. This allows the user to perform "what if" exercises, as well as software development, while decreasing development time and effort.

The component constructors are based on special parts called meta-parts. These parts are the blueprint for the generated Ada code. They facilitate requirements input and contain all necessary construction information for the development of the Ada code.

Twelve component constructors were developed in CAMP-2 -- Kalman Filter, Finite State Machine, Pitch Autopilot, Lateral/Directional Autopilot, Navigation Subsystem, Navigation Component, Data Bus Interface, Data Type, Task Shell, Time-Driven Sequencer, Event-Driven Sequencer and Process Controller.

PES Environment

In CAMP-2 the prototype PES was developed on a Symbolics 3620 computer (the PES was later moved to a VAX in CAMP-3). This machine is a single-user LISP workstation designed to support the LISP programming language. An expert system shell was used as the foundation for the PES. The system was developed using ART (Automated Reasoning Tool from Inference, Corp.) and Common LISP. This environment was selected for several reasons, paramount of which was the availability of a production quality expert system shell. At the time, ART was the most mature system available

on the market. This consequently mandated the selection of the processor; ART was only available on the LISP processor.

11th Missile Demonstration

One of the goals in CAMP-2 was to use the CAMP software in its intended domain in the most realistic situation possible. The true test of the CAMP parts and the PES came when they were used to build the 11th missile.

A cruise missile system originally implemented in JOVIAL J73 was selected as the 11th Missile. This application utilized MIL-STD-1750A (hereafter referred to as 1750A) processors and a MIL-STD-1553B data bus. Navigation, guidance and support functions of the 11th Missile were re-implemented using the parts and the PES to gauge the productivity improvement associated with both.

To measure exclusive productivity increases associated with the CAMP parts and the PES, two versions of the 11th Missile were written. Version one was written using parts without the assistance of the PES and was referred to as the parts method. Version two, called the PES method, used the PES and the parts to generate and unit test the Kalman filter code for the system. The re-implementation of the 11th Missile revealed impressive productivity results.

11th Missile Re-implementation Results

The results of the parts method evaluation were very promising. A productivity increase of 15% was observed implementing the 11th Missile using only the parts (without the assistance of the PES). In other words, a development team would save 15% of their efforts if they were to implement the 11th Missile using the CAMP parts instead of developing from scratch. The parts accounted for 18.1% of the total 11th Missile.

The PES method resulted in an convincing 28% improvement in productivity using the PES Kalman filter constructor. The constructor generated code or instantiated parts to develop 70.1% of the Kalman filter component.

Another benefit of the 11th Missile application is the demonstration of Ada used in a RTE application. At the time of the evaluation, Ada was criticized for not being suitable for this domain. The 11th Missile was developed using approximately 21,000 lines of Ada code and only 21 lines of assembly code. It worked admirably and is well suited for RTE applications.

ARMONICS Benchmarks

A benchmark suite was also developed during CAMP-2 to measure the efficiency of compilers for suitability for programming armonics (armament electronics) software. The benchmarks also facilitated the evaluation of the CAMP parts. The suite contains benchmarks designed to gauge compilation and run-time performance.

The CAMP compilation benchmarks determine the ability of an Ada compiler to compile and link complex Ada syntax and semantics typically found in the CAMP parts and armonic software. The applicability of these benchmarks is not limited to the parts; the benchmarks could be used for other domains as well.

The execution benchmarks include mathematical functions and typical use missile applications such as guidance, navigation, and Kalman filtering as benchmarks. Run-time data such as execution time and output are produced by these benchmarks. Code size is also determined.

CAMP-3: Technology Transition

The third and final CAMP phase, CAMP-3, began in July 1988 and will end in September 1991. The CAMP-3 goals are to refine and transition the technology demonstrated in CAMP-2. More specifically, CAMP-3 involves parts maintenance and enhancement, PES re-engineering, meta-constructor development, and various technology transition efforts.

Parts Maintenance

The primary goal of parts maintenance was to correct possible errors discovered during the 11th Missile application. Also, since the CAMP parts have been distributed to nearly 300 agencies at their request, these agencies were viewed as a valuable evaluation source. A questionnaire was distributed to all recipients of the CAMP parts requesting ideas for corrections, enhancements and modifications. Meetings were also conducted to solicit these inputs from McDonnell Douglas sister organizations. Only one error in the CAMP parts was reported.

Fifty-one new parts were identified, designed, coded and added to the parts set as a result of this maintenance effort. In addition, existing parts were enhanced to make the CAMP parts more robust. Consequently, the total number of parts increased to over 500.

PES Catalog Re-engineering

The PES re-engineering goal was to develop a robust, all-Ada, production-quality version of the PES. (While the meta-constructor is included in the PES, it is still regarded as not production quality.) As previously mentioned, the prototype PES was implemented on a Symbolics LISP machine using the ART expert system shell. While this is an excellent environment for rapid development and prototyping of a PES, it is not suitable for wide-scale distribution of a production-quality PES. The Symbolics environment is very specialized relying on system dependencies for successful PES operation which is a detriment to software reuse. To maximize software reuse, the delivery environment must be accessible to several organizations. This eliminated a Symbolics delivery system; few potential CAMP users had access to this machine. Therefore, an Ada/microvax platform was selected. All PES software is being re-engineered to Ada thus exploiting Ada's portability for maximum distribution. (Of course the parts

were already written in Ada.) System dependencies were isolated and site tailorable features were added. All third-party software was removed to eliminate the need for software licenses.

Meta-constructor

The component constructors developed during CAMP-2 proved the feasibility of such Ada-producing capabilities. However, these constructors are very costly to develop and maintain. Therefore, a meta-constructor is being developed during CAMP-3 to alleviate these problems. A meta-constructor is a constructor designed to produce other component constructors. This approach will lower the overall cost of developing constructors and, ultimately, tailored Ada code.

Technology Transition

Perhaps the most important responsibility of a program manager is technology transition. A successful program will benefit no one if it is not made available to the intended user. CAMP's ultimate users are software engineers and weapon systems developers. Therefore, the CAMP-3 program embarked on an aggressive technology transition campaign which included the development and distribution of a CAMP brochure, a reuse manual and a videotape, as well as a demonstration of CAMP in action at a national conference -- Tri-Ada '90. The most significant technology transition effort was the actual distribution of the CAMP parts and catalog as previously discussed.

A brochure was also developed to "spread the word" about CAMP. It explains the entire CAMP effort and contains a complete listing of all products including the actual parts and documentation. A partial list is provided at the end of this paper. The brochure's greatest asset is information on how to obtain these products. This brochure is available from either the McDonnell Douglas CAMP program manager, (314) 232-0278, or the USAF CAMP program manager, (904) 882-8264.

One of the most popular commodities is a manual entitled "Developing and Using Ada Parts in Real-Time Embedded Applications." The manual embodies the overall CAMP experience and includes informative techniques and methods for developing and using Ada parts in the real-time embedded realm. The manual is available from the Defense Technical Information Center (DTIC) by ordering document number AFATL-TR-90-67.

An executive overview videotape was also produced to describe Ada issues, software reuse issues and how the CAMP program attempts to alleviate them. The videotape has been distributed to nearly 70 organizations to heighten awareness of the CAMP approach to the software crisis.

Conclusions

CAMP has been a pathfinding program in several respects. It demonstrated that software reuse is feasible and valuable and that Ada can effectively handle real-time embedded applications. Throughout the CAMP program, one theme emerged: Software reuse and the development of software parts must be precisely planned. An ad hoc approach to software reuse is destined to failure.

The future of software reuse is bright. CAMP took a tremendous step towards institutionalizing software reuse as a standard way of doing business. Indeed, industry must rely more and more on software reuse to remain competitive in this era of austere budgets.

Acknowledgements

The author thanks Constance Palmer and the McDonnell Douglas CAMP team as well as Chris Anderson for their thoughtful review and comment of this paper. Their ideas and time are truly appreciated.

Other Readings/Materials

The following CAMP documents are available through the Defense Technical Information Center:

Developing and Using Ada Parts in Real-Time Embedded Applications: A manual that gives guidance in how to develop and use reusable software. Order AFATL-TR-90-67.

CAMP-1 Final Technical Report: Three volumes covering domain analysis, parts specification, parts composition system study. Order AFATL-TR-85-93, Volumes 1-3.

CAMP-2 Final Technical Report: Three volumes covering parts and parts composition system development, 11th Missile Application development, and Armonics Benchmarks development. Order AFATL-TR-88-62, Volumes 1-3.

The CAMP software products listed below are available through the Data & Analysis Center for Software, P.O. Box 120, Utica, New York 13503; the telephone number is (315) 336-0937.

CAMP Ada Parts: ANSI standard tapes containing source code for the parts, test code and utilities, and design documents in machine readable form.

Parts Engineering System (PES) Catalog Version 1.1: ANSI standard tapes containing the CAMP catalog and the data needed to load it with the CAMP parts. This is in Ada, uses no commercial third-party software, and runs under VAX/VMS.

Benchmark Tape: An ANSI standard tape containing the benchmarks, standard data files, and VAX command procedures for executing the benchmarks on VAX hardware.

Development and Verification of Software for Flight Safety Critical Systems

by

H. Afzali and Dr. A. Mattissek

LITEF GmbH
Lörracher Straße, D7800 Freiburg, Germany

1. Summary

In Flight Safety Critical Systems where the lives of people and/or mission success is depending on, errors in the Computer Software Components can have a catastrophic impact on the safety.

The requirements for the software development and maintenance of Flight Safety Critical systems differ in some aspects from the systems which do not fall into this category. The reason for these requirement is to produce "the right product" at the very beginning of the system's usage and to ensure special attention is paid throughout the whole service life of the equipment.

The reliability and safety requirements can reach a point where testing alone is not sufficient. Consequently adequate control mechanisms have to be applied. The software configuration management, quality control, verification and validation must be rigorously adhered to.

For the development of the equipment software, a set of development standards and additional procedures for the implementation of Safety Critical Functions are defined.

LITEF applied the standards and procedures for the development of the Inertial Measurement Unit which is a part of the Flight Control System and Seat Sequencer Unit which is part of the Ejection Seat.

In this paper, some critical technology needs are described for supporting the development and verification process of such systems and the activities which have to be performed during the development phases for identifying, assessing and eliminating or minimizing hazards in a systematic way.

2. Introduction

In recent years, software has gradually been given more and more responsibility. Today the software has complete control over many Safety Critical Functions on some air vehicles. In fact, it would be impossible for a human to manually fly several of the modern aircrafts. This is because they require complex control inputs at faster than human speeds in order to prevent loss of control leading to a crash. However the question persists. Are we able to verify that the software is sufficiently free from errors which would have a catastrophic impact on the safety?.

It would be fair to ask the questions: "what are the risks incurred by using the software?" and "what is the probability that the plane will crash due to a fault in the software?". From the point of view of the pilot, who doesn't care if the problem is the hardware or the software, the issue may be rephrased and put into the bigger context. "Given that I am going to fly that airplane on a one-hour mission today, what are my chances of returning safely?".

3. Knowledge Base

Originally each software system will be considered as "unsafe". This label can be only removed and replaced by a "safe" label after sufficient knowledge about its safety status exists.

A newly designed software for which there is none or little knowledge in the way of analysis and test results can not be considered safe.

On the other hand, it could happen that the entire analysis and testing of software reveals no need for changes. At the end of the qualification of the software which was originally labelled "unsafe", it is determined to be "safe", even though absolutely no changes were made. The only thing that has changed since the initial design release was our knowledge about the entity in question.

As the figure above illustrates, the quality of the established standards and procedures, methods and tools, prepared documentation, review reports, the results of the verification, testing and software safety analysis will impact the decision making process related to the safety of the software.
Not to forget the engineers who participate in the development of the project.

4. Development Standards

A structured development philosophy and Verification and Validation approach is particularly important in the case of Flight Safety Critical Systems. Much documentation exists relative to standards, procedures, methods, tools and environment which support this type of development. Standards and guidelines are described in a sophisticated way and must be applied to the related projects. The efforts required during the development process increases with the criticality of the application.

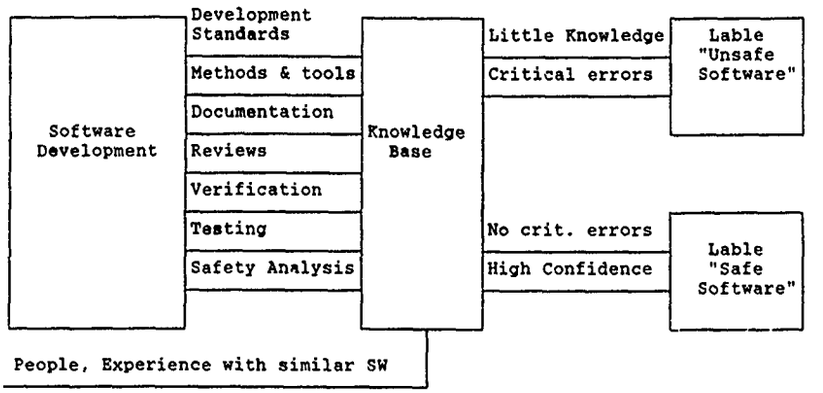


Figure 1 Knowledge Base

For each of the five major category of activities which are :

- Software project management
- Software configuration management
- Software quality evaluation
- Software engineering
- Software testing

detailed tasks are defined. Model texts will help the engineers to prepare the documentation in a standard way.

In order to minimize the very costly and most difficult to detect errors in the early phases of the Software projects, Methods and tools Application Standards, Design and Coding Standards are specified to support effectively the production of the software. In the requirements analysis phase, the design phases, and the coding phase.
In addition development guidelines and procedures for software safety tasks are established.

5. Safety Analysis Activities

The objectives of the safety analysis tasks are :

- -to identify the hazards
- -to eliminate the hazards if possible through design or reducing the associated risks to an acceptable level
- -to minimize the hazardous events

The analysis will result in reports, recommendations, guidelines and corrective actions.

The fundamental principles to achieve a high degree of flight safety are:

- -System Level Management
- -Flight Safety Analysis
- -Information Flow

The Flight Safety Program is a top level system engineering activity which integrates flight safety concept and analysis into all phases of the project. The Software Safety Analysis is performed in parallel to other development activities and is accomplished from the system level down to the component level. It is a step by-step procedure which attempts to exhaustively identify all potential hazards to which the system or its functions, could be subjected to. It is fundamentaly a top-down approach which goes as deep as necessary in order to adequately describe the hazards with respect to their consequences on flight safety. The hazard analysis is structured into a hierachy.

In order to manage and coordinate the flight safety related activities, a flight safety engineer with enough experience and thorough knowledge about the application, hardware, software, shall be appointed to the project.

In order to perform a comprehensive analysis, the flight safety engineer must collect and organize all information related to the software development process. The relevant data to the flight safety is focussed and used in the flight safety analysis process.

Based on the preliminary list of potential hazards and given the system's operational environment, the system safety engineer develops hazard scenarios. A scenario is the possible sequences of events and circumstances that can lead to a consequence. For each event involved in the hazard scenario, the system safety engineer considers failure modes that can lead to these events.

Different techniques are known for the accomplishment of the hazard analysis. LITEF's approach for identification of the hazards in the Software requirements analysis phase and the preliminary design phase is the usage of the 'Fault Tree' technique . Based on critical signals defined in the specification, the critical functions in the Software Requirements Specification and associated Computer Software Components (CSC) in the 'Software Top Level Design Document' are identified.

Critical functions and interfaces are identified and allocated to the system components. Multiple systems with dissimilar software, Backup systems, redundant configurations shall be considered in the overall design so that flight safety failures can only occur as a result of multiple failures.

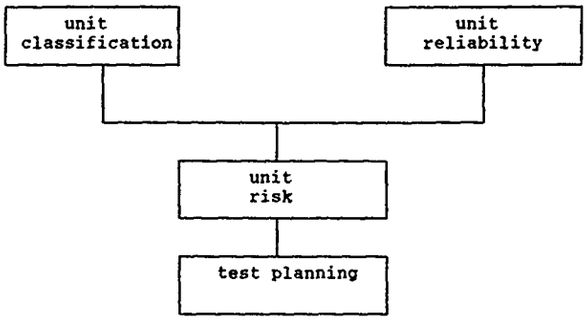


Figure 2 Planning model

The analysis continues in the software detailed design, coding and unit testing phase.

The objectives of the tasks during the detailed design phase are :

- -to identify potential failures and define their effects on safety
- -to identify necessary design changes
- -to identify the extensiveness required for testing and V&V activities

In order to complement the system safety tasks and reaching the objectives defined for software safety, a Failure Mode, Effects and Criticality Analysis (FMECA) is performed in the detailed design phase. The level of analysis in this phase is the computer software unit (CSU) level. These units are identified in the detailed design document. A CSU is examined to determine its impact on the reliability and safety. Functions and units which are identified as critical during FMECA will undergo a more extensive testing.

Following steps are performed for FMECA :

- -Planning
- -Analysis
- -Reporting

The details of the analysis approach, documentation and worksheets, report formats, interfaces and other analyses performed at the code level are defined in the FMECA plan.

Failure severity category and hazard consequence severity category are assigned to each computer software unit.

Static and Dynamic code analysis are performed to the source code. The objectve of Static code analysis is to identify the deficiencies in the data flow, control flow and information flow and to assess the complexity of the programs.

The objective of dynamic code analysis is to verify that the test cases for the units provide sufficient coverage of the source code.

These activities are performed in accordance with the Mil-STD-882B task series 300.

6. Safety testing

Testing is generally performed until the test engineers feel confident that the software is reliable enough and can be released. Various testing and reliability models has been developed to determine the level of reliability. Normally these models does not address the failure impact in the case of safecy critical systems. The amount of testing required is heavily dependent on the potential effects of failure on the flight safety.

In order to define the specific safety testing requirements in the functional, component and unit level of the software, LITEF uses the results of the flight safety analysis, statistical data of the previous projects and the on-going test results.

Unit risk is estimated based on the data described. The tests are planned proportionally to the risk of each unit.

Units are classified by relating the units to the hazards and their consequences. In the detailed design phase all units are analyzed to identify those which use or update the data related to the critical functions. The units are classified according to their impacts on the safety.

In addition the expected number of faults per unit is estimated. This estimation is based on the complexity of the unit and the statistical data which are collected during past years for units of similar projects. The a priori distribution for the unit reliability is formulated for each unit after it has been coded.
The unit classification and unit reliability model both will serve as a qualitative assessment of the unit risk and consequently a better planning of test efforts.

During the unit test phase and subsequent phases, the unit fault statistics are collected. The unit fault rates used for the updates of the a priori distribution and consequently the test plan.

7. Application

The Seat sequencer software is categorized as flight safety critical.
It is a microprocessor based unit which controls the timing of the various seat sub-systems (e.g. drogue canister catapult

firing, parachute container catapult firing). Control of these timings will be based on information provided by seat sequencer mounted sensors which establish the ejection conditions of acceleration, base pressure and dynamic pressure. The sequencer operates independently of the aircraft.

The system architecture constitutes the following principles :

- -triple redundant microprocessor channels
- -redundant sensing of environmental conditions
- -harmonization of intermediate results between channels
- -2-of-3 H/W voting before Electro Explosive Devices(EED)-ignition

The basic design philosophy is to completely eliminate single point failures by building a triple redundant system.

Each seat sequencer channel contains the same program. Each channel samples the environmental data during the ejection. The intermediate results of each microprocessor channel is harmonized with both neighbouring channels. The EED ignition decision of each channel is passed to a hardware voter which in turn makes a 2-of-3 voting for a final ignition decision.

The organization of the software is very simple given the critical nature of the application. It implies the repetitive execution of tasks within a predefined period of time.

In order to facilitate the target testing of the units and the Computer Software Components (CSC's), consideration has been made in the overall software design. Each CSU or CSC can be isolated from the other parts of the software by external commands and tested by downloading the individual data of different test cases.

The sequencer is considered to be flight safety critical for, if the sequencer fails to fire the pyrotechnics in the correct sequence and at the correct time.

Based on this top level hazard, the system is analyzed and the sub events which can lead to the top event are identified. This analysis are continued until the basic events are reached. The fault tree technique has been used for this analysis. Based on the safety analysis, a report is prepared and recommendations have been given for the re-design or as requirements for the subsequent activities.

7. Conclusion

A software development methodology has been presented to produce highly reliable software for flight safety critical applications.

In addition a test planning method for this type of applications has been presented.

References

(1) Military Standard, Defence System Software Development DOD-STD-2167

(2) Military Standard, System Safety Program Requirements MIL-STD-882B

(3) Software Considerations in Airborne systems and Equipment Certification RTCA/DO-178A

(4) Software Entwicklungsstandard der Bundeswehr, Vorgehensmodell

(5) EFA Standards (A comprehensive set of standards and procedures)

(6) S.Sherer, Methodology for the Assessment of Software Risk, Doctoral dissertation, Wharton School, Univ. of Pennsylvania, Philadelphia

(7) J.D.Musa, A.Iannino, and K. Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw Hill, New York, 1987

(8) S.Sherer, "Measuring the Risk of Software Failure: A Financial Application,"

(9) Halverson, LITEF's Flight Safety Program With Respect To The Software, Philosophical Background and Practical Implementation

(10) H. Afzali, W. Hassenpflug, Development and Verification of Software for Flight Safety Critical Strapdown Systems

REPORT DOCUMENTATION PAGE			
1. Recipient's Reference	**2. Originator's Reference**	**3. Further Reference**	**4. Security Classification of Document**
	AGARD-CP-503	ISBN 92-835-0629-4	UNCLASSIFIED

5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France
6. Title	SOFTWARE FOR GUIDANCE AND CONTROL
7. Presented at	the Guidance and Control Panel 52nd Symposium held at the Helexpo, Thessaloniki, Greece, from 7th May to 10th May 1991.

8. Author(s)/Editor(s)	**9. Date**
Various	September 1991
10. Author's/Editor's Address	**11. Pages**
Various	254

12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the back covers of all AGARD publications.

13. Keywords/Descriptors

Ada	Expert systems
Safe Ada	Rapid prototyping
Object oriented design	Software generators
Formal specifications	Real-time software
Fourth generation languages	Flight critical software
Reusable software	Software design methods
Transformational methods	

14. Abstract

This volume contains the 23 unclassified papers, presented at the Guidance and Control Panel Symposium held at the Helexpo, Thessaloniki, Greece from 7th to 9th May 1991.

The papers were presented covering the following headings:

— Tools and methods from a user's viewpoint,
— General requirements on software;
— Integrated programmes support environments;
— Software requirements;
— Design methods for real-time software;
— ADA applications;
— Automated software generation approaches.

AGARD Conference Proceedings 503 Advisory Group for Aerospace Research and Development, NATO SOFTWARE FOR GUIDANCE AND CONTROL Published September 1991 254 pages This volume contains the 23 unclassified papers, presented at the Guidance and Control Panel Symposium held at the Helexpo, Thessaloniki, Greece from 7th to 9th May 1991. The papers were presented covering the following headings — Tools and methods from a user's viewpoint. PTO.	AGARD-CP-503 Ada Safe Ada Object oriented design Formal specifications Fourth generation languages Reusable software Transformational methods Expert systems Rapid prototyping Software generators Real-time software Flight critical software Software design methods	AGARD Conference Proceedings 503 Advisory Group for Aerospace Research and Development, NATO SOFTWARE FOR GUIDANCE AND CONTROL Published September 1991 254 pages This volume contains the 23 unclassified papers, presented at the Guidance and Control Panel Symposium held at the Helexpo, Thessaloniki, Greece from 7th to 9th May 1991. The papers were presented covering the following headings — Tools and methods from a user's viewpoint. PTO.	AGARD-CP-503 Ada Safe Ada Object oriented design Formal specifications Fourth generation languages Reusable software Transformational methods Expert systems Rapid prototyping Software generators Real-time software Flight critical software Software design methods
AGARD Conference Proceedings 503 Advisory Group for Aerospace Research and Development, NATO SOFTWARE FOR GUIDANCE AND CONTROL Published September 1991 254 pages This volume contains the 23 unclassified papers, presented at the Guidance and Control Panel Symposium held at the Helexpo, Thessaloniki, Greece from 7th to 9th May 1991. The papers were presented covering the following headings: — Tools and methods from a user's viewpoint: PTO	AGARD-CP-503 Ada Safe Ada Object oriented design Formal specifications Fourth generation languages Reusable software Transformational methods Expert systems Rapid prototyping Software generators Real-time software Flight critical software Software design methods	AGARD Conference Proceedings 503 Advisory Group for Aerospace Research and Development, NATO SOFTWARE FOR GUIDANCE AND CONTROL Published September 1991 254 pages This volume contains the 23 unclassified papers, presented at the Guidance and Control Panel Symposium held at the Helexpo, Thessaloniki, Greece from 7th to 9th May 1991 The papers were presented covering the following headings — Tools and methods from a user's viewpoint: P.T.O.	AGARD-CP-503 Ada Safe Ada Object oriented design Formal specifications Fourth generation languages Reusable software Transformational methods Expert systems Rapid prototyping Software generators Real-time software Flight critical software Software design methods

— General requirements on software, — Integrated programmes support environments, — Software requirements; — Design methods for real-time software, -- ADA applications; — Automated software generation approaches ISBN 92-835 0629-4	— General requirements on software; — Integrated programmes support environments; — Software requirements, — Design methods for real-time software; — ADA applications; — Automated software generation approaches, ISBN 92-835-0629-4
— General requirements on software, — Integrated programmes support environments; — Software requirements; — Design methods for real-time software; — ADA applications; — Automated software generation approaches. ISBN 92-835-0629-4	— General requirements on software; — Integrated programmes support environments; — Software requirements, — Design methods for real-time software; — ADA applications; — Automated software generation approaches ISBN 92-835-0629-4

AGARD

NATO OTAN

7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE

FRANCE

Téléphone (1)47.38.57.00 · Télex 610 176
Télécopie (1)47.38.57.99

DIFFUSION DES PUBLICATIONS AGARD NON CLASSIFIEES

L'AGARD ne détient pas de stocks de ses publications, dans un but de distribution générale à l'adresse ci-dessus. La diffusion initiale des publications de l'AGARD est effectuée auprès des pays membres de cette organisation par l'intermédiaire des Centres Nationaux de Distribution suivants. A l'exception des Etats-Unis, ces centres disposent parfois d'exemplaires additionnels; dans les cas contraire, on peut se procurer ces exemplaires sous forme de microfiches ou de microcopies auprès des Agences de Vente dont la liste suite.

CENTRES DE DIFFUSION NATIONAUX

ALLEMAGNE
Fachinformationszentrum,
Karlsruhe
D-7514 Eggenstein-Leopoldshafen 2

BELGIQUE
Coordonnateur AGARD-VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Directeur du Service des Renseignements Scientifiques
Ministère de la Défense Nationale
Ottawa, Ontario K1A 0K2

DANEMARK
Danish Defence Research Board
Ved Idraetsparken 4
2100 Copenhagen Ø

ESPAGNE
INTA (AGARD Publications)
Pintor Rosales 34
28008 Madrid

ETATS-UNIS
National Aeronautics and Space Administration
Langley Research Center
M/S 180
Hampton, Virginia 23665

FRANCE
O.N.E.R.A. (Direction)
29, Avenue de la Division Leclerc
92320, Châtillon sous Bagneux

GRECE
Hellenic Air Force
Air War College
Scientific and Technical Library
Dekelia Air Force Base
Dekelia, Athens TGA 1010

ISLANDE
Director of Aviation
c/o Flugrad
Reykjavik

ITALIE
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
Aeroporto Pratica di Mare
00040 Pomezia (Roma)

LUXEMBOURG
Voir Belgique

NORVEGE
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
N-2007 Kjeller

PAYS-BAS
Netherlands Delegation to AGARD
National Aerospace Laboratory NLR
Kluyverweg 1
2629 HS Delft

PORTUGAL
Portuguese National Coordinator to AGARD
Gabinete de Estudos e Programas
CLAFA
Base de Alfragide
Alfragide
2700 Amadora

ROYAUME UNI
Defence Research Information Centre
Kentigern House
65 Brown Street
Glasgow G2 8EX

TURQUIE
Milli Savunma Başkanlığı (MSB)
ARGE Daire Başkanlığı (ARGE)
Ankara

LE CENTRE NATIONAL DE DISTRIBUTION DES ETATS-UNIS (NASA) NE DETIENT PAS DE STOCKS DES PUBLICATIONS AGARD ET LES DEMANDES D'EXEMPLAIRES DOIVENT ETRE ADRESSEES DIRECTEMENT AU SERVICE NATIONAL TECHNIQUE DE L'INFORMATION (NTIS) DONT L'ADRESSE SUIT.

AGENCES DE VENTE

National Technical Information Service
(NTIS)
5285 Port Royal Road
Springfield, Virginia 22161
Etats-Unis

ESA/Information Retrieval Service
European Space Agency
10, rue Mario Nikis
75015 Paris
France

The British Library
Document Supply Division
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
Royaume Uni

Les demandes de microfiches ou de photocopies de documents AGARD (y compris les demandes faites auprès du NTIS) doivent comporter la dénomination AGARD, ainsi que le numéro de série de l'AGARD (par exemple AGARD-AG-315). Des informations analogues, telles que le titre et la date de publication sont souhaitables. Veuiller noter qu'il y a lieu de spécifier AGARD-R-nnn et AGARD-AR-nnn lors de la commande de rapports AGARD et des rapports consultatifs AGARD respectivement. Des références bibliographiques complètes ainsi que des résumés des publications AGARD figurent dans les journaux suivants:

Scientifique and Technical Aerospace Reports (STAR)
publié par la NASA Scientific and Technical
Information Division
NASA Headquarters (NTT)
Washington D.C. 20546
Etats-Unis

Government Reports Announcements and Index (GRA&I)
publié par le National Technical Information Service
Springfield
Virginia 22161
Etats-Unis

(accessible également en mode interactif dans la base de données bibliographiques en ligne du NTIS, et sur CD-ROM)

Imprimé par Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

AGARD

NATO · OTAN

7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE

FRANCE

Telephone (1)47.38.57.00 · Telex 610 176
Telefax (1)47.38.57.99

DISTRIBUTION OF UNCLASSIFIED AGARD PUBLICATIONS

AGARD does NOT hold stocks of AGARD publications at the above address for general distribution. Initial distribution of AGARD publications is made to AGARD Member Nations through the following National Distribution Centres. Further copies are sometimes available from these Centres (except in the United States), but if not may be purchased in Microfiche or Photocopy form from the Sales Agencies listed below.

NATIONAL DISTRIBUTION CENTRES

BELGIUM
Coordonnateur AGARD — VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Director Scientific Information Services
Dept of National Defence
Ottawa, Ontario K1A 0K2

DENMARK

FRAN
(
2
9

GERM
F
K
D

GREEC
He
Ai
Sci
De
Del

ICELAN
Dir
c/o.
Reyk,. ...

ITALY
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
Aeroporto Pratica di Mare
00040 Pomezia (Roma)

LUXEMBOURG
See Belgium

NETHERLANDS
Netherlands Delegation to AGARD
National Aerospace Laboratory, NLR
Kluyverweg 1
2629 HS Delft

NORWAY

Establishment

or to AGARD
is

...formation Centre
Kentigern House
65 Brown Street
Glasgow G2 8EX

UNITED STATES
National Aeronautics and Space Administration (NASA)
Langley Research Center
M/S 180
Hampton, Virginia 23665

THE UNITED STATES NATIONAL DISTRIBUTION CENTRE (NASA) DOES NOT HOLD STOCKS OF AGARD PUBLICATIONS, AND APPLICATIONS FOR COPIES SHOULD BE MADE DIRECT TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS) AT THE ADDRESS BELOW.

SALES AGENCIES

National Technical
Information Service (NTIS)
5285 Port Royal Road
Springfield, Virginia 22161
United States

ESA/Information Retrieval Service
European Space Agency
10, rue Mario Nikis
75015 Paris
France

The British Library
Document Supply Centre
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
United Kingdom

Requests for microfiches or photocopies of AGARD documents (including requests to NTIS) should include the word 'AGARD' and the AGARD serial number (for example AGARD-AG-315). Collateral information such as title and publication date is desirable. Note that AGARD Reports and Advisory Reports should be specified as AGARD-R-nnn and AGARD-AR-nnn, respectively. Full bibliographical references and abstracts of AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR)
published by NASA Scientific and Technical
Information Division
NASA Headquarters (NTT)
Washington D.C. 20546
United States

Government Reports Announcements and Index (GRA&I)
published by the National Technical Information Service
Springfield
Virginia 22161
United States
(also available online in the NTIS Bibliographic
Database or on CD-ROM)

Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

ISBN 92-835-0629-4